Bonus Chapter

Sprite Interaction: A Physics Primer

in this chapter

- Using sprite interactions for games
- Using sprites as a control interfaces for game control
- Simulating physics for your application

IN THIS CHAPTER, I don't show you how to build a complete or pretty application, but you do learn some advanced uses for sprites and App Inventor animation components. Although the blocks figures in this chapter allow you to completely build the apps, they are not fully functional applications so much as concept demonstrations.

You have seen how a canvas and sprites are related in the AlphaDroid application. Sprites and other animation components such as the Ball sprite work only in the context of a canvas. In this chapter, you discover two different methods for simulating realistic motion in your App Inventor apps.

Examining Different Techniques for Realistic Motion

I demonstrate the first method with a simple Breakout-style clone app.

NOTE Breakout was one of the earliest computer games after Pong. To play, you had to hit a ball using a paddle into a series of stacked blocks, thus destroying the blocks without losing your ball off the bottom of the screen. In this chapter, I show you how to create this effect in App Inventor.

In this application, whenever a sprite collides with an object, you simply negate the value of the *heading* (another word for *direction*). In other words, when a collision occurs, the heading of the sprite is reversed. Now, this roughly approaches the behavior of two objects when a collision occurs, but in reality, the physics at work when a collision occurs between two objects is much more complex. Simulating actual physics between two objects is a very complex computation. Even approximating acceleration, gravity, inertia, friction, and speed requires a great deal of complex mathematics. Simply changing the heading of the object, however, gives you a rough approximation of what happens in a real collision. The `.Bounce` method for ImageSprite's does something similar. Using a combination of `.Bounce` and the `negate` block to reverse the sprite's heading gives a usable approximation of real-life physics.

The second method is a lot more complex than the first and uses far more of that inexplicable sort of math that your high school teachers swore would be useful later in life. The second example is a core physics engine with no real application to demonstrate it (although there will be block illustrations to make the physics engine useful). The second demonstration uses Kinematics, a branch of mechanics that allows the description and modeling of motion with mathematical formulas.

NOTE

I won't take the time to explain all the formulae used in this example. Not only would it take an entire book, but I barely understand the math involved. But don't worry: You don't have to understand it either to get some benefit from it. Instead, I present the second method of simulating physical object interaction as an advanced example to be used as the core of more complex applications. You can find many books and Web sites (start at `www.physicsclassroom.com/class/1dkin/u1l6a.cfm`) with excellent detailed material on Kinematics and physics modeling.

After you build up the second method of physics modeling, you have the blocks and the core to build your own games and applications around.

Figure BC-1 shows the BreakDroid design sketch.

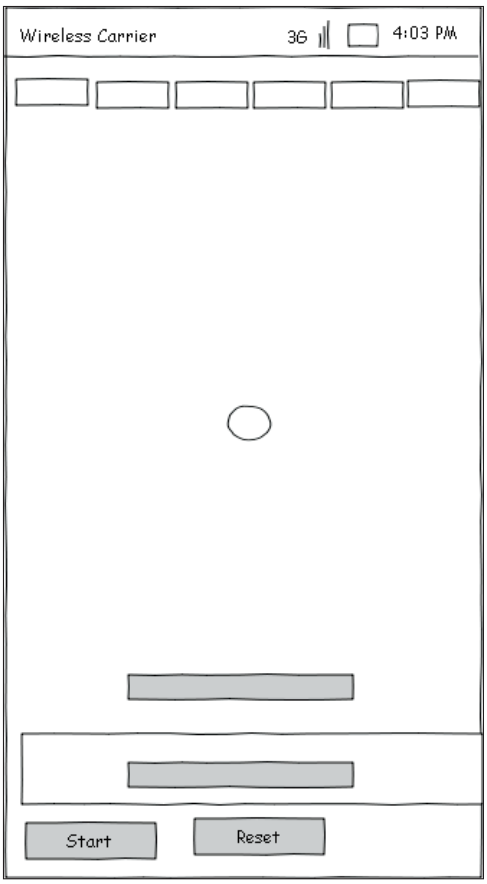

FIGURE BC-1: The design sketch

Your primitives

The design for BreakDroid is very basic and provides a framework for demonstrating the basic physics of a bouncing ball. It also allows you to see sprite interaction and how you can use that interaction to create games or other user interface interactions. The blocks interact with the Ball sprite by disappearing. I leave scoring, sound, and other elements up to you for a challenge.

- A row of sprite blocks
- A Ball sprite
- A method to handle interaction between colliding sprites
- A Paddle sprite the user can interact with
- A method to keep track of sprite collisions
- A method to reset the game interface
- A method to handle bouncing when two sprites collide
- A method to handle selective edge bounces

New blocks

These are the new blocks I introduce in this project:

- `Negate`
- `Collide With`

New components

The Ball component is the only new component for this project. The Ball component is simply a sprite from the Animation palette that is shaped like a ball. It has no special properties or characteristics that distinguish it from a sprite:

- Ball

Your progression

These are the logical steps to accomplish your design goals and assemble the BreakDroid application:

1. Place the game play canvas.
2. Place the paddle movement canvas.
3. Place the block sprites and the ball sprite.
4. Place the Start and Reset buttons.
5. Build a scorekeeping procedure.
6. Handle collision events.
7. Handle edge events.
8. Handle the Start button event.
9. Build a game reset procedure.
10. Handle the Reset button event.

Getting Started on BreakDroid

The interface for the BreakDroid application is minimal, but it gives you a sandbox to play around in to explore sprite interaction. You use two canvas components for the game play interface. The first is the canvas for the block sprites, paddle, and ball. The second is used as a control interface to control the paddle. You use the `Screen1.Title` to display score information much as you did with the TwiTorial project.

1. Start a new project and name it `BreakDroid`.
2. Set the Icon with the icon file from the project files that you downloaded from the companion Web site.
3. Uncheck the `Scrollable` property.

Next, place all of the Canvas and Sprite components for the game play interface.

1. Drag and drop a Canvas component onto the viewer. Set its `Height` and `Width` property to `Fill Parent`.
2. Drag and drop six ImageSprites from the Animation palette onto the canvas. Align them across the top of the canvas, as shown in Figure BC-1.
3. Rename the ImageSprites `sprtBlock1` through `sprtBlock6`. You should end up with six sprites with sequential names. These are the blocks that disappear when the Ball sprite collides with them.

4. Upload the block.png file from the project files into the Media column so it can be applied to your sprites.

   If you need instructions on downloading the project files from the companion Web site, see the Introduction to this book.

5. Set the `Image` property of each `sprtBlock` to the block.png.

6. Drag and drop a Ball component in the center of the canvas.

7. Drag and drop a new ImageSprite at the bottom of the canvas. Set the component name to sprtPaddle. Set its `Image` property to the paddle.png file from the project files. This is the paddle that interacts with the ball. The Control paddle you will add moves this paddle to catch the ball and bounce it back toward the blocks.

8. Drag and drop a new Canvas component below the Canvas1 component.

9. Set the new Canvas2 `Width` property to `Fill Parent`. Set the `Height` property to `40` pixels.

10. Set the `BackgroundColor` property to `Grey`. This gives the second canvas some visual separation.

11. Drag and drop a new image sprite into the second canvas. Rename it `sprtControlPaddle`.

    This is a control mechanism for the paddle. If the user has to drag the paddle with a direct tap, their finger blocks the screen. You use a "ghost" paddle whose actions the real paddle mirrors.

12. Apply the paddle.png image from the project files just as you did with the main paddle, to `sprtControlPaddle`.

Next, add your two interface buttons. You have a Start button to allow the user to start the ball moving. You also place a Reset button to completely reset the game interface and allow the user to start over. As I mentioned previously, this is just a framework and would require other processes and some retooling to be a workable game.

1. Drag and drop a HorizontalArrangement below the second Canvas component.

2. Drag and drop a button into the HorizontalArrangement. Change the Text property to `Start` and rename it `btnStart` in the Component column.

3. Drag and drop a second button into the HorizontalArrangement and rename it `btnReset`. Set the `Text` property to `Reset`.

Those are all of your basic interface elements for the BreakDroid application. Most of the fun begins in the Blocks Editor.

The BreakDroid application interface should look like Figure BC-2.

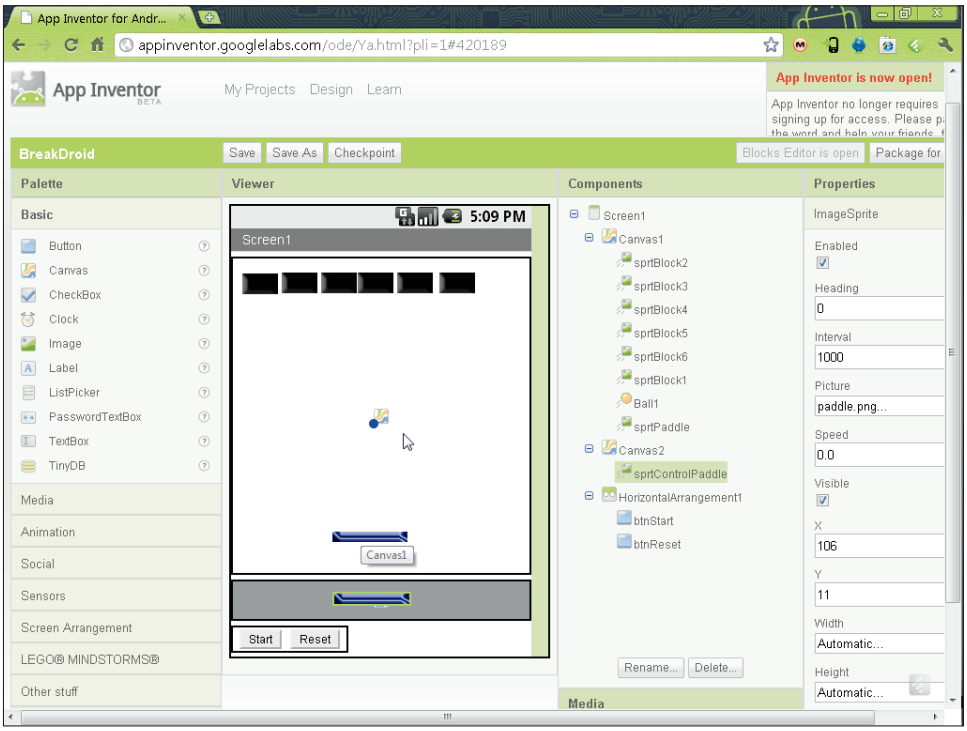

FIGURE BC-2: The BreakDroid user interface

Start adding the logic and sprite interaction by following these steps:

1. Open the Blocks Editor and typeblock the `Screen1.Initialize` event handler.
2. Typeblock the `Screen1.Title [to]` block and snap it into the event handler.
3. Typeblock a `text` block and set the text to `Press start and use lowest paddle to move game paddle`. This gives the user some indication of how to utilize your interface. You later change the `Screen1Title` to display the user's score.

Each time the Ball sprite collides with a block sprite, you need to increment a score variable, update a score display, and make the block disappear. This is one rudimentary way you could handle scoring or collision events. Depending on your game type, collision events might need

to be recorded and checked. For instance, you might want to record the number of times a sprite is hit and then check to see if that number is high enough to consider the sprite "dead."

In the following steps, you create a procedure called `procScoreIncrement`. You call this procedure each time a block/ball collision occurs.

For this case, a simple incrementing of the variable and display in the `Screen1.Title` is sufficient:

1. Define a new variable and change its name to `varScore`.
2. Typeblock a `0` number block and snap it into the score variable.
3. Typeblock a new procedure and name it `procScoreIncrement`.
4. Typeblock the `varScore [to]` block and snap it into `procScoreIncrement`.
5. Typeblock an addition (+) block. Snap it into the `varScore` block.
6. Typeblock the `varScore` global variable block and snap it into the first socket on the addition (+) operator.
7. Typeblock a numeral `1` number block and snap it into the second open socket on the addition (+) operator.
8. Typeblock the `Screen1.Title [to]` block and snap it into the `procScoreIncrement` procedure next.
9. Typeblock a `join` block and snap it into the `Screen1` block.
10. Typeblock a `text` block and change the text to `BreakDroid Score`. Snap it into the first socket on the `join` block.
11. Typeblock the `varScore` global variable block and snap it into the second socket on the `join` block.

    Every time the `procScoreIncrement` procedure is called, the `varScore` variable is incremented and the score display in the Screen title bar is updated.

NOTE Currently the title bar cannot be removed. Using it as a display method for data is a smart use of screen area.

You also need to call a reset procedure to reset the ball when it misses the paddle. This includes returning the ball to a given X/Y position and resetting its heading and speed:

1. Typeblock a new procedure and set its name to `procBallReset`.
2. Typeblock the `Ball1.X [to]` block and snap it into the procedure.
3. Typeblock a numeral `143` number block and snap it into the `X` position block.
4. Typeblock the `Ball1.Y [to]` block and snap it in to the procedure.
5. Typeblock a numeral `169` block and snap it into the `Y` position block.
6. Typeblock the `Ball1Heading [to]` block and snap it in next.
7. Typeblock a numeral `0` number block and snap it into the `Heading` block.
8. Typeblock the `Ball1. Speed [to]` block and snap it in next.
9. Typeblock a numeral `0` number block and snap it into the speed block.

Whenever the `procBallReset` procedure is called, it returns the ball to the given X/Y coordinates and removes its speed and heading, effectively freezing the ball on the screen. The Start button then reinitializes the ball's heading and speed.

When the ball collides with a block sprite, not only should it increment the score; it should also bounce off. This is an opportunity to use the `negate` block. When the `collide` event occurs, you call a procedure called `procBounce` that simply negates the ball's current heading.

The `negate` block is a mathematic block that takes an integer and returns the exact opposite in numerical value; for example, `180` becomes `-180`. When used in conjunction with a ball's current heading, it can be used to reverse the ball's heading. This gives the appearance of a bounce:

1. Typeblock a new procedure and name it `procBounce`.
2. Typeblock the `Ball1.Heading [to]` block and snap it into the procedure.
3. Typeblock the `negate` block and snap it into the `Ball1.Heading` block.
4. Typeblock the `Ball1.Heading` block and snap it into the `negate` block.

After you have the `procScoreIncrement` and `procBallReset` and `procBounce` created, you can handle all of the collision events for your sprites. The desired behavior for a collision between the ball sprite and a block sprite is that the block sprite should disappear, the score should be incremented, and the ball should to appear to bounce. You build each of the `.CollidedWith` event handlers in the same way. Each `sprtBlock` sprite generates an event when another sprite collides with it. You use this event to generate your desired behavior:

1. Drag and drop the `sprtBlock1.CollidedWith` event handler.
2. Typeblock the `procScoreIncrement` procedure `call` block and snap it into the event.
3. Typeblock the `procBounce` procedure call and snap it into the event handler.
4. Next, typeblock the `.Visible` block for the sprite you are working with. In this case, typeblock the `sprtBlock1.Visible [to]` block and snap it into the block.
5. Set the `.Visible` block with a `false` block.
6. Repeat Steps 1-5 for each of the `.CollidedWith` event handlers for each of the sprite blocks.

When you finish setting up the `.CollidedWith` event handlers, you should have six `.CollidedWith` event handlers, like the one in Figure BC-3.

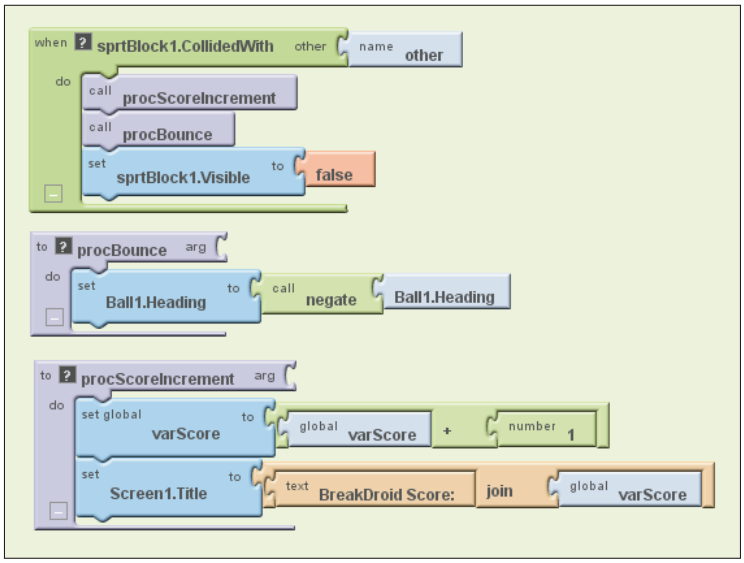

FIGURE BC-3: The .CollidedWith event with the procBounce and procScore Increment procedures

The paddle control method you implement in the following steps is an important exercise in using canvas drag or sprite drag event X/Y coordinates as input for other sprite actions. When the user drags the `sprtControlPaddle` back and forth on Canvas2, the X coordinate moves the game paddle, `sprtPaddle`, on the game canvas. The logic for this is that the user's finger blocks the paddle if the user has to click directly on the paddle. This is not the only way the finger block issue could be addressed; however, it's important to understand that the information that comes from sprite/canvas interaction can be used anywhere in your application to effect or change user interaction.

1. Drag and drop the `sprtControlPaddle.Dragged` event handler from the sprtControlPaddle blocks drawer. This is one of those enormous event handlers with bunches of parameters. You are only interested in the `CurrentX` parameter. This parameter contains the sprites X coordinates whenever it is dragged on a canvas. You use the `CurrentX` event handler to set the X position of the game play paddle and the control paddle whenever the control paddle position changes.
2. Typeblock the `sprtPaddle.X [to]` and snap it into the event handler.
3. Typeblock the `currentX VALUE` block and snap it into the `sprtPaddle.X` block.
4. Typeblock the `sprtControlPaddle.X [to]` and snap it into the event handler next.
5. Typeblock the `currentX` again and snap it into the `sprtControlPaddle` block.

Now whenever the user moves the control paddle by dragging it, the `sprtControl-Paddle.Dragged` event occurs and the position of the two paddles is updated to the dragged X location. A control sprite is very useful for designing games. You can use it create thumb joysticks buttons or other control structures.

You need to handle the event when the ball collides with the game paddle. As with all sprite collisions, this generates an event. You use this event to call the `procBounce` event:

1. Drag and drop the `sprtPaddle.CollidedWith` event handler from the sprtPaddle blocks drawer.
2. Typeblock the `procBounce` procedure `call` block and snap it into the event handler.

This is all that is needed to bounce the ball back towards the blocks.

The primary goal of a block-breaker type of game is to keep the ball from running off the bottom of the screen and remove the blocks. At this point, you have handled making the blocks disappear, but you still need to handle what happens when the ball reaches an edge. When a sprite reaches the edge of a canvas, the `.EdgeReached` event is generated and you can use this event to determine what should be done. The event has a parameter with it that contains the particular edge that has been reached. The edges of a canvas are numbered (see the AlphaDroid project in Chapter 6 for more). The top edge is numbered as 1 and the bottom edge is numbered –1. You need logic that asks, "What edge has the ball sprite reached? And what should happen when the edge is –1."

1. Typeblock the `Ball1.EdgeReached` event handler.
2. Typeblock an `IfElse` block and snap it into the event handler.

3. Typeblock an equals (=) comparison operator and snap it into the `test` socket on the `IfElse` block.
4. Typeblock the `edge` value parameter block from the `.EdgeReached` event handler and snap it into the first socket on the comparison operator.
5. Typeblock a numeral `–1` number block and snap it into the second socket on the comparison operator. Make sure that your number is a –1 number as the bottom edge of the canvas is represented by –1.
6. Typeblock the `procBallReset` procedure call and snap it into the `then-do` socket on the `IfElse` block. If the edge reached is –1, the ball will be reset.
7. Typeblock the `Ball1.Bounce` block and snap it into the `else-do` socket on the `IfElse` block.
8. Typeblock the `edge` value parameter block and snap it into the `Ball1.Bounce` block.

Now if the test returns as untrue, the ball calls its native `.Bounce` function. The `.Bounce` function is, to all intents and purposes, the same as your `procBounce` procedure.

The only events left to handle are the two button events. The Start button event gives the ball sprite an initial heading and speed:

1. Typeblock the `btnStart.Click` event handler.
2. Typeblock the `Ball1.Heading [to]` and snap it into the event handler.
3. Typeblock a numeral `250` number block and snap it into the `.Heading` block.
4. Typeblock the `Ball1.Speed [to]` block and snap it next into the event handler.
5. Typeblock a numeral `10` number block and snap it into the `.Speed` block.

The user tapping the `btnStart.Click` starts the ball moving towards the bottom edge and slightly to the left.

The only event left is the Reset button. The Reset button makes all the blocks come back and resets the score. It also resets the ball with the `procBallReset` procedure.

1. Typeblock the `btnReset.Click` event handler.
2. Typeblock the `varScore [to]` block and snap it into the event handler. Populate it with a numeral `0` number block. This resets the `score` variable.

3. Typeblock the `Screen1.Title [to]` block and snap it next into the event handler.
4. Typeblock a `join` block and snap it into the `.Title` block.
5. Typeblock a `text` block and change the text to `BreakDroid Score:`. Snap it into the first open socket on the `join` block.
6. Typeblock the `varScore` global variable block and snap it into the second socket on the `join` block.

Next you return all the `sprtBlock`'s Visible property to `true`. When a game has finished in your version of this game, all of the block sprites will be invisible. For a new game to be available, you need to return all the blocks to visible.

1. Typeblock the `sprtBlock1.Visible [to]` block and snap it in next in the event handler.
2. Typeblock a `true` block and snap it into the `.Visible` block.
3. Repeat Steps 1-2 for all the sprtBlocks. You end up with each of the sprtBlocks being made visible, as shown in Figure BC-4.

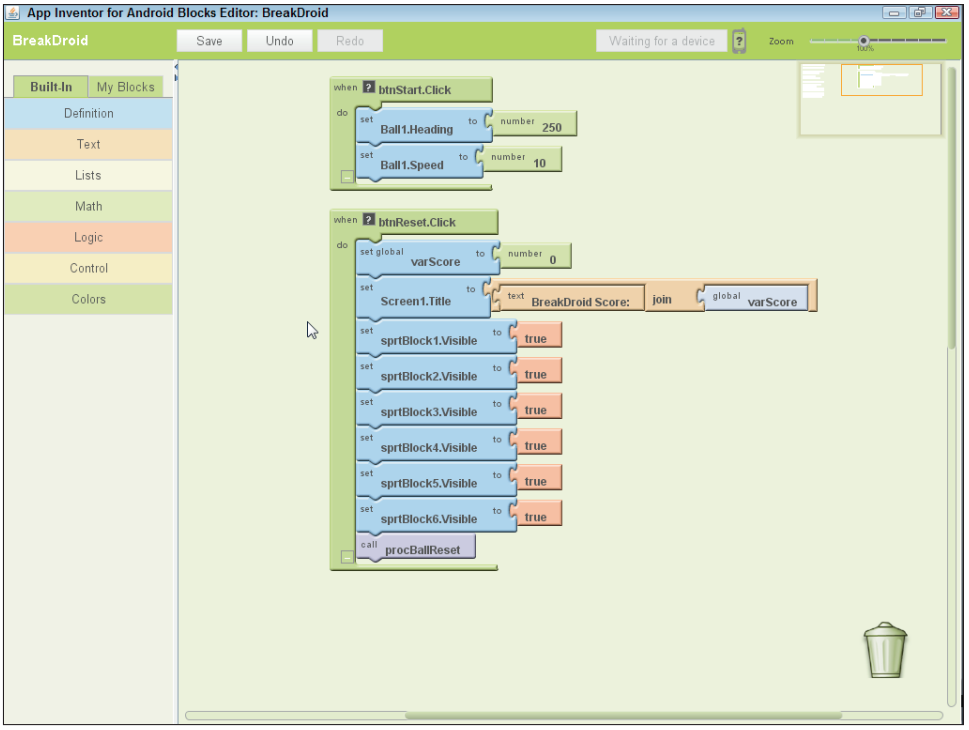

FIGURE BC-4: The Reset button and Start button .Click event handlers

4. Typeblock the `procBallReset` procedure call and snap it into the event handler beneath the last `.Visible` block.

All of the possible events have been handled for your BreakDroid application. Package and download it to your phone and start breaking some blocks.(Refer to Chapter 1 to refresh on how to package and install.) Obviously, this is not currently a complete game application. Try some of the following challenges to make a complete game:

- Limited lives (reaching the bottom edge of the canvas costs one life).
- Multiple levels without using multiple screens. (Think about starting with lots of invisible sprites that are enabled as the game progresses.)
- Sounds for bounce events.
- A "win" scenario.

Creating a Pseudo-Physics Engine

I'm not going to show you how to build an actual application with this sample physics engine. However, a sample use application is included with the chapter project files you downloaded from the companion Web site. You can upload the .zip source file into your projects and see it in action.

NOTE I don't attempt to explain the complex math involved with the Kinematics of this physics engine. Josh Turner on the Google App Inventor forums did the core logic and blocks programming for the physics example in his Physics Demonstration app and I owe him thanks for allowing me to use it here. The end result is a simulation of the interaction between two objects with a rough approximation of both inertia and gravity. This is done by constantly updating the heading, speed, and coordinates of a sprite based on a series of defined constants according to some rather complex math.

You need a timer to constantly update the speed, heading, and position settings. In the given example application, the Ball sprite is given some constants to approximate motion. However, in a real application, those constants would likely be variable based on user input or sprite interactions.

Your design

Figure BC-5 is a picture of the core blocks and formulae that make up the kinematics engine for simulating physics for your sprites in your applications.

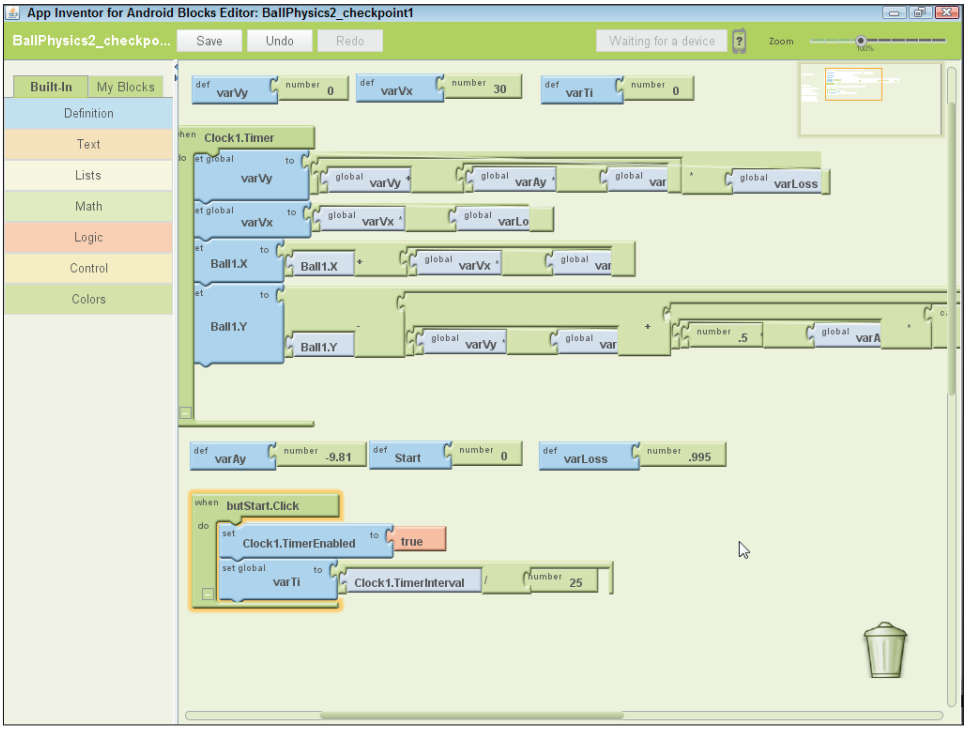

FIGURE BC-5: The completed Kinematics engine blocks

Your primitives

These are the basic goals of a physics engine such as the Kinematics engine. It strives to simulate

- Object interaction
- Inertial effects
- Gravitational pull
- The changes to speed and heading over time based on the previous items

Your progression

Actually building an application is far more involved than what you will do in this simple demonstration. However, these are the steps to build the quasi-physics core:

1. Define variables for X, Y, time, and gravity/loss
2. Create a method for regularly checking and updating the heading and speed of a sprite.

Getting Started on the Physics Engine

You need a skeleton application consisting of a canvas and a Ball sprite or a sprite and a Clock component. The steps in the process are for building the physics engine; applying it in your application is up to your imagination.

After you have your canvas and Ball sprite and any sprites you want the ball to interact with ready, proceed with building the kinematics engine. First, define all of the following variables:

- `varTi` with an initial value of the Clock1.Timer interval divided by 25:

```
(.TimerInterval/25)
```

NOTE In the included sample application, the `varTi` is initialized with the Start button event. This is to allow for playing with the `.TimerInterval` property on the `Clock1.Timer`. If you want to change the load on the processor and accuracy of the formulae, you can change the `Clock1.TimerInterval` either up or down. If you want, however, you can declare varTi as a constant.

- `varVy` with an initial value of `0 This is the initial Y axis speed.`
- `varVx` with an initial value of `30`. This is the initial X axis speed. (This initial value is just to give the ball some initial sideways movement for interesting bounces.)
- `varLoss` with an initial value of `.995`. This is a value that indication loss of mostion or decaky.
- `varAy` with an initial value of `-9.81`. This is a value indicating initial acceleration.

The last two constants represent the external forces being applied to the object or sprite.

After you have the variables defined, start building the `Clock1.Timer` event, which should be set to `5` milliseconds for smooth heading transitions with high processor usage, or to `15` milliseconds to make it a little easier on the device processor.

The first two series of blocks set the `Vy` and the `Vx` variables, which are later be used to change the X/Y coordinates of your sprite. The formula for the Vy variable is

```
(Vy + (Ay*Ti)) * varLoss
```

TIP

Refer to `www.physicsclassroom.com/class/1dkin/u1l6a.cfm` for a primer on the equation.

Using Vy plus the formula allows you to use external forces or settings at any point in the clock cycles:

1. Typeblock the `Clock1.Timer` event handler.
2. Typeblock the `varVy [to] block` and snap it into the event handler.
3. Typeblock a multiplication operator `(×)` and snap it into the `varVy` block.
4. Typeblock the `varLoss` global value block and snap it into the second socket on the multiplication operator `(×)`.
5. Typeblock an addition operator (+) and snap it into the first socket on the multiplication operator `(×)`.
6. Typeblock the `varVy` global value block and snap it into the first socket on the addition operator (+) block.
7. Typeblock a second multiplication operator (×) block and snap it into the second socket on the addition operator (+).
8. Typeblock the `varAy` global value block and snap it into the first socket on the second nested multiplication operator (×).
9. Typeblock the `varTi` global value block and snap it into the second socket on the second nested multiplication operator (×).

The completed `varVy` blocks in the `Clock1.Timer` should look like Figure BC-6.

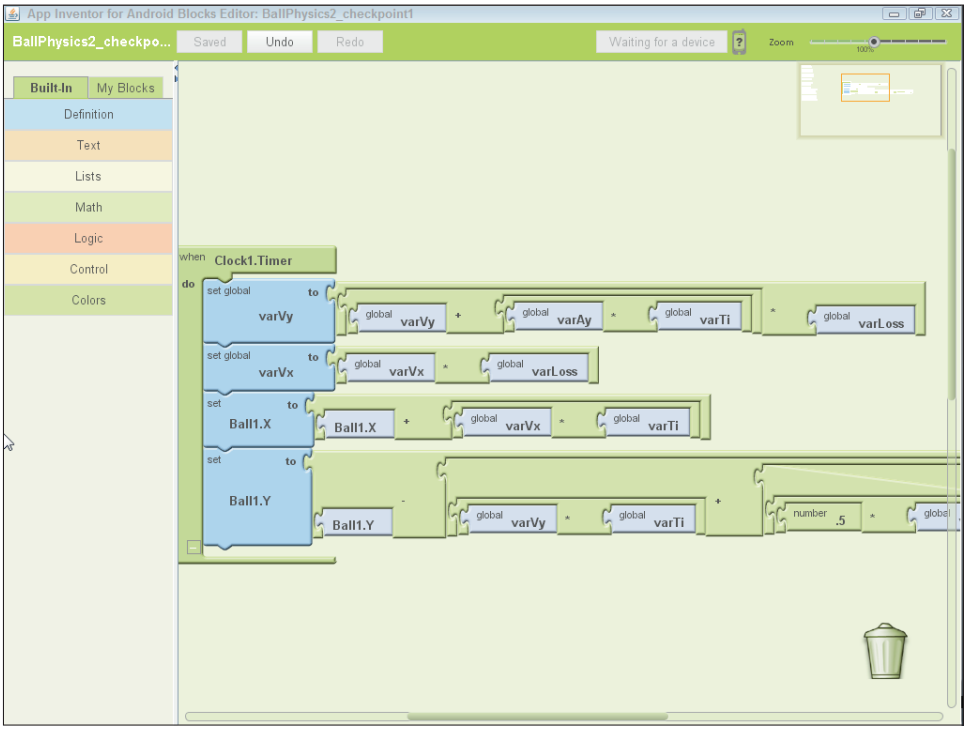

FIGURE BC-6: The varVy formulae blocks

Next set the `varVx` variable, for which the formula for `X` is significantly easier:

```
Vx * varLoss
```

1. Typeblock the `varVx [to]` block and snap in below the `varVy` blocks.
2. Typeblock a multiplication operator (×) block and snap it into the `varVx` block.
3. Typeblock the `varVx` global value block and snap it into the first socket on the multiplication operator (×).
4. Typeblock the `varLoss` global value block and snap it into the second socket on the multiplication operator (×).

Your `varVx` block should look like the one in Figure BC-7.

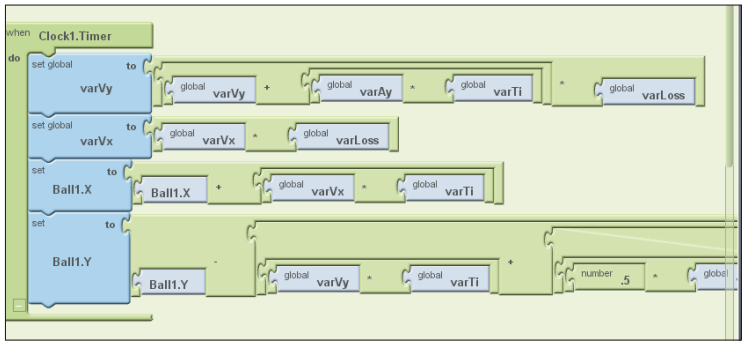

FIGURE BC-7: The completed formulae blocks for the varVx variable

Each time the clock processes, your sprite's coordinates are adjusted based on the contents of the variables.

The X coordinates formula uses the contents of the `varVx` variable and is fairly simple:

```
Ball1.X = Ball1.X + (Vx * Ti)
```

Build the formulae into the `Clock1.Timer` event. You set the X coordinates of the Ball1 sprite. The Ball1 sprite would be replaced by your own sprite from your own application when you use this physics engine.

1. Typeblock the `Ball1.X [to]` block and snap it in below the `varVx` series of blocks in the `Clock1.Timer` event.
2. Typeblock an addition operator (+) block and snap it into the `Ball1.X` block.
3. Typeblock the `Ball1.X` value block and snap it into the first socket on the addition operator (+).
4. Typeblock a multiplication operator (×) and snap it into the second socket on the addition operator (+).
5. Typeblock the `varVx` global value block and snap it into the first socket on the multiplication operator (×) nested in the addition operator (+).
6. Typeblock the `varTi` global value block and snap it into the second socket on the multiplication operator (×).

The completed `Ball1.X [to]` block should look like Figure BC-8.

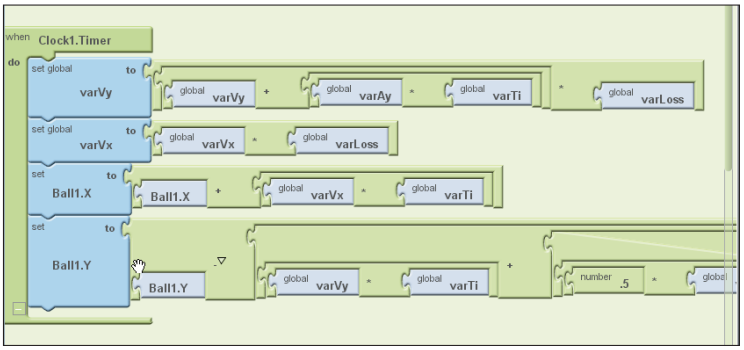

FIGURE BC-8: The completed X coordinate blocks

The Y coordinates have a more complex formula, with many nested math blocks, as you move through building the `Ball1.Y` blocks. If you get lost, refer to Figure BC-9:

1. Typeblock the `Ball1.Y [to]` block and snap it in below the `Ball1.X` blocks.
2. Typeblock a minus operator (–) block and snap it into the `Ball1.Y` block.
3. Typeblock the `Ball1.Y` value report block and snap it into the first socket on the minus operator (–) block.
4. Typeblock an addition operator (+) block and snap it into the minus operator (–) block second socket.

The next three steps apply only to the first socket on the addition operator (+) you just placed:

1. Typeblock a multiplication operator (–) and snap it in.
2. Typeblock the `varVy` global variable value block and snap it into the first socket on the addition operator (+).
3. Typeblock the `varTi` global variable value block and snap it in the second socket on the addition operator (+).

The next few steps apply to the second socket on the addition operator (+) that is socketed in the main minus operator (–):

1. Typeblock a multiplication operator (×) and snap it into the socket.
2. Typeblock a second multiplication operator (×) and snap it into the first socket on the first multiplication operator.

3. Typeblock a `.5` numeral number block and snap it into the first socket on the nested multiplication operator (×).
4. Typeblock the `varAy` global value block and snap it into the second socket on the nested multiplication operator (×).
5. Typeblock a `expt` math block and snap it into the second socket on the outside multiplication operator (×).

   The `expt` operator returns the result of raising the number plugged into the `base` socket by the number snapped into the exponent socket.
6. Typeblock the `varTi` global value block and snap it into the `base` socket on the `expt` block.
7. Typeblock a numeral 2 number block and snap it into the `exponent` socket.

The entire `Ball1.Y` block series should look like Figure BC-9.

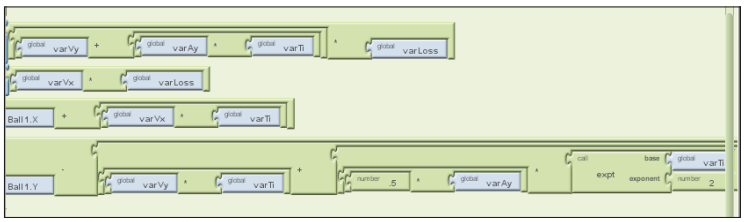

FIGURE BC-9: The completed Y coordinate blocks

> The `Ball1` blocks would be replaced with the blocks from whatever sprite you were working with in your application. **REMEMBER**

If you built a sandbox app for this engine, you can have the sprite, in this case, the Ball1, bounce realistically by applying the `negate` block to the appropriate coordinate variable. If the object that your sprite should bounce from is vertical, you would negate the `varVy` variable. If the object is horizontal, you would negate the `varVx` variable. You can see an application of this in the sample application. In the `.CollidedWith` event, you call the `varVy [to]` or `varVx [to]` block with the `negate` block and then the `variable` value block. You can see an example of this in Figure BC-10. In the figure, you can see a bounce from a horizontal and a bounce from a vertical object.

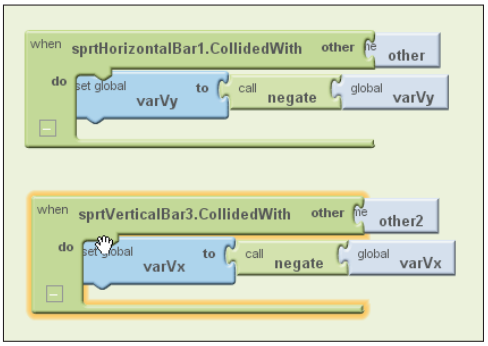

FIGURE BC-10: The physics engine applied for collide events

The same is true when your sprite reaches an edge. For horizontal edges, you negate the `varVy`; for vertical edges, you negate the `varVx` variable. You can see the `Ball1.EdgeReached` event set up to use the physics engine for edge events in Figure BC-11.

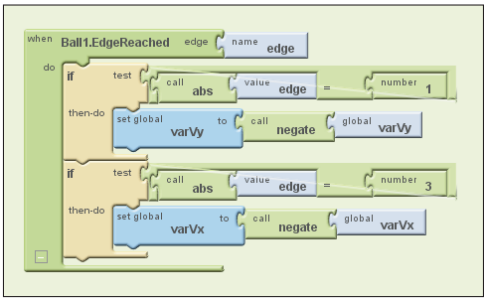

FIGURE BC-11: The physics engine applied to edge events

It should be noted that this physics engine is very processor-intensive and will likely run fairly slowly on older devices. Most modern 1GHz or greater processors should handle this method fairly well. The best way to know if your application will work on a certain device is to package and load the application on the device.

Challenges

The pseudo-physics engine demonstrated here has many possible derivations and expansions. The simplest way to achieve the sprite behavior you desire is to change both the constant and the variables a little at a time and see how the sprite behaves. The emulator or a connected device is the best way to test these kinds of incremental changes.

Try some of the following challenges:

- Implement the physics engine with the previous BreakDroid application

NOTE

Set the Paddle `collide` event to increase the acceleration (that is, the speed) of the ball.

- Emulate the moon's or Jupiter's gravity
- Implement directional wind