Android Game Development with AppInventor

by

Anshul Bhagi
S.B EECS MIT 2011

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 23, 2012

Author:

Anshul Bhagi
Department of Electrical Engineering and Computer Science

Certified
by:

Prof. Hal Abelson
Class of 1992 Professor of Computer Science and Engineering
MIT Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted
by:

Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Android Game Development with AppInventor

Abstract

Anshul Bhagi[1]
Department of Electrical Engineering and Computer Science

AppInventor is an educational learning tool provided by MIT that allows users to build Android apps without any knowledge of programming. As AppInventor gains popularity amongst educators and students around the world, it will become increasingly important to ensure that the tool offers its users the breadth and depth of app-development functionality they desire. In anticipation of AppInventor's expanding role and influence in educational institutions worldwide (middle schools and high-schools, primarily), this thesis focuses on the age group of 3rd to 12th grade students, and on the topic that is of greatest interest to them: gaming, animation, and graphics.

The aim of this thesis is to identify AppInventor's existing capabilities and limitations with respect to game development, and to implement ideas (both pedagogical and technological in nature) that will improve the diversity, complexity, aesthetic appeal, and performance of games that can be built using AppInventor. The author of this thesis believes that if AppInventor's game development capabilities can be augmented, the adoption rate of the tool and its popularity amongst school students will be impacted very positively.

In this thesis, the author describes his personal experiences teaching AppInventor game development in India and USA, as well as the limitations (in teaching methodology and in AppInventor's feature set) that he identified through this experience. The author's primary contributions are the development of a hands-on curriculum for a 40-hour AppInventor Game Development course, and the implementation of several new features and components for AppInventor. The author will be traveling to China and India in Summer 2012 to test to what extent his creative curriculum and novel AppInventor modifications facilitate the development of games using AppInventor.

[1] abhagi@mit.edu

Table of Contents

Chapter 1

Introduction & Overview

About AppInventor.

AppInventor is a web-based tool developed jointly by MIT and Google to enable the creation of Android apps via a visual, block-based development environment that requires no prior knowledge of programming languages. Since its conception, AppInventor has gained popularity amongst educators as a learning tool for fundamental concepts in computer science, and more importantly, as a medium through which students can exercise creativity and practice innovation. AppInventor (along with its peer applications at MIT such as Scratch and StarLogo) has empowered a whole new age group (specifically, elementary and middle school students) to create applications for android phones -- a task that had previously been impossible due to the numerous demands and requirements of the Android app creation process (knowledge of Java, comprehension of computer science principles, and familiarity with software development tools and environments such as Eclipse, the Android SDK, Android Developer Bridge, etc.).

Why AppInventor should support game development.

Ever since AppInventor found a new home at MIT's Center for Mobile Learning in early 2012 (it was previously housed at Google), it has been receiving significant attention from educational communities that hope to use the tool to facilitate technology learning in the classroom and beyond. Much of AppInventor's current user base is teachers in high-schools and colleges, and as the tool continues to attract more users, it is likely that many of these users will be teachers of students in grades 3 thru 12 (i.e. students who are old enough to know how to use computers and design/develop apps, yet not experienced enough to get their hands dirty with the Android SDK). Students of this age group are quite fond of gaming, and so we anticipate that there will be a strong demand for developing graphically appealing, interactive single-player and multi-player games using App Inventor in the coming years.

It is therefore important that the AppInventor team at MIT prepare for the imminent growth of the AppInventor game development community. Accordingly, this thesis looks at where AppInventor currently stands with respect to game development and how its game development capabilities can be improved and extended.

Game development with AppInventor today.

Developing games with AppInventor's existing feature set is very much a possibility, and in fact the process is quite easy to learn, thanks to AppInventor's intuitive interface and the numerous tutorials (e.g. PaintPot, QuizMe, MoleMash) available online at beta.appinventor.mit.edu/learn/tutorials.

In particular, the following AppInventor components prove quite useful for the task of developing games:

Buttons:
These are perhaps the simplest way in which a game developer can get input from the game player. Buttons can be used to create a joystick (with controls for moving up, down, left, right), and to perform any other actions in response to a button click.

Canvas:
Canvas components are useful for multiple reasons. First, they are the component where game objects are drawn -- shapes such as rectangles, ovals, etc. can be drawn on these using the methods provided by AppInventor in the Blocks Editor, and images can be uploaded as backgrounds. Second, they are containers for other components known as sprites (described below), and provide a set of bounds within which sprite objects can be drawn and moved around. Third, Canvas objects can be used as a source of touch-input, and they provide a "dragged" and "touched" event that the app developer can respond to. Fourth, these objects can be used as a spacing mechanism in layouts, e.g. if a developer wishes to center a button on the screen, he/she can add an empty/transparent canvas on each side within a Horizontal Layout, and set the width property of each canvas to "fill parent" -- magically, the end result will be the centering of the button or whatever other object is between the two outside canvases.

<u>Clock:</u>
Clocks are instrumental for game development since they are the time-keepers. They have an Interval property which the game developer can define and modify at any time, and they fire an event after each interval expires. The game developer can easily react to the firing of these clock timer events in the Blocks Editor in order to perform certain tasks (such as updating the location of a Sprite) every few milliseconds.

<u>Sound:</u>
Sound effects are easy to add in App Inventor and can make a significant impact on perceived game quality. The 'Sound' component is an invisible component that allows the developer to easily select a source sound file and to easily play this file in response to other actions from the Blocks Editor.

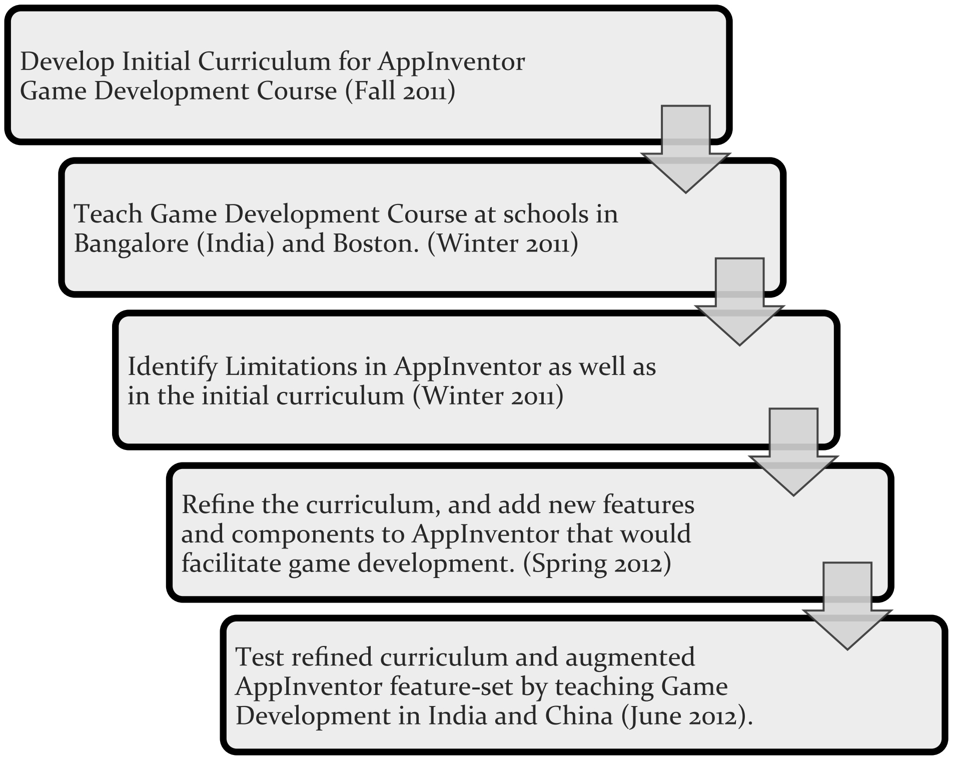
<u>Ball:</u>
The ball component is perfect for games that actually require balls that need to move around on a canvas. The ball component supports rectangle-based collision detection and implements the physics of ball movement/bouncing. The game developer can provide it a direction (called a heading), a speed, a radius, and the time interval the ball should wait between each pair of movements.

<u>ImageSprite:</u>
Sprites are a fundamental part of any game that requires animation or motion. They are essentially more generic 'ball' components in that they can be visualized by any image that the developer uploads (animated gifs not supported). They support rectangle-based collision detection and are restricted to movement within the bounds of the canvas in which they reside (they can't go off-screen).

<u>GameClient:</u>
This component is the foundation for building multi-player games with AppInventor. It allows for web-based communication between multiple clients (phones) and a single game server that the game developer deploys independently.

Author's approach to improving AppInventor's game development capabilities.

The goal of this thesis is to identify AppInventor's existing capabilities and limitations with respect to game development, and to implement ideas (both pedagogical and technological in nature) that will improve the diversity, complexity, aesthetic appeal, and performance of games that can be built using AppInventor. The author adopted the following iterative hands-on approach to understanding how AppInventor could be improved for game development:


Develop Initial Curriculum for AppInventor Game Development Course (Fall 2011)
Teach Game Development Course at schools in Bangalore (India) and Boston. (Winter 2011)
Identify Limitations in AppInventor as well as in the initial curriculum (Winter 2011)
Refine the curriculum, and add new features and components to AppInventor that would facilitate game development. (Spring 2012)
Test refined curriculum and augmented AppInventor feature-set by teaching Game Development in India and China (June 2012).

The remainder of this report describes each step of this process in detail. Chapter 2 describes the original curriculum; Chapter 3 communicates the experience of teaching AppInventor in India and Boston and describes the learning that emerged from this experience; Chapters 4 and 5 address modifications to AppInventor and to the curriculum, respectively; Chapter 6 addresses future plans and suggests additional features that should be added to AppInventor, and Chapter 7 summarizes the contributions of this work.

Chapter 2

Original Curriculum

Curriculum Overview

	Planned Activities	Concepts Learned
Day 1 (8 hrs)	Ice Breakers Playing with Android Phones AppInventor Setup Overview of AppInventor's interface Project 1: HelloPurr and Media Player Project 2: Calculator	Sound, Button, Screen, event handling, calling procedures, connecting to device, Player, screen arrangements, Label, TextBox, getting and setting property values, if-else statements, packaging apps to phone
Day 2 (8 hrs)	Project 3: Drawing Program Project 4: Balls, Sprites, and Motion of Objects 4 ways to make things move (automatically; using Clock, using Buttons, using Accelerometer)	Canvas, handling 'dragged' and 'touched' events, drawing lines and circles, colors, Ball, ImageSprite, making objects move, edge-detection, bouncing ball, defining custom variables, defining custom procedures, Accelerometer
Day 3 (8 hrs)	Project 5: Car on a Road Project 6: Paper Prototyping Begin final project (game of student's choice)	collision detection, random-number generation, making multiple objects move simultaneously using Clock,
Day 4 (8 hrs)	Continue final project (game of student's choice)	Instruction on Day 4 will be one-on-one and highly customized to each student's or group's needs.
Day 5 (8 hrs)	Publishing AppInventor apps to Google Play Continue final project (game of student's choice) Presentations of final projects	Google Play developer registration process, downloading AppInventor apps as .apk files, uploading .apk files to Google Play. The remainder of the learning on Day 5 will be customized to each student's or group's needs.

Day 1

Begin with ice breakers (nicknames, fun facts, what you want to learn, why you enrolled in course, etc.)

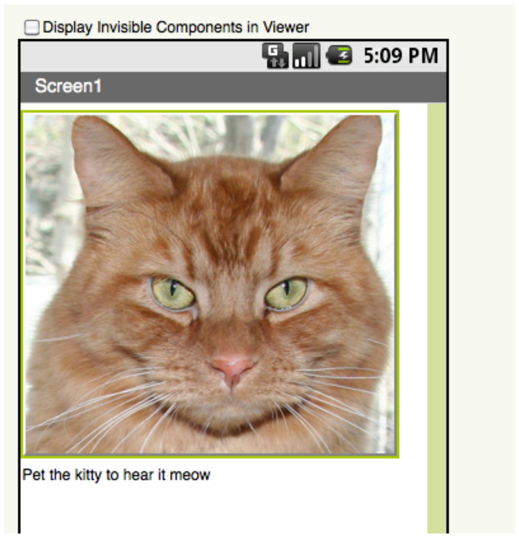
Hand out phones and connector cables, allow students to play around with and get a feel for the devices. Attract their attention to the Google Play application (formerly known as the Market) on the Android phones and excite them about the prospect of publishing their own application to Google Play (to be discussed on Day 5).

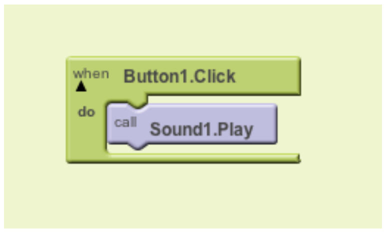
Ask students to perform AppInventor setup, following the instructions at http://beta.appinventor.mit.edu/learn/setup/.

Walk them through the basics of AppInventor's interface (designer view, palette, properties panel, visible vs. non-visible components, and blocks editor).

Project 1: HelloPurr and MediaPlayer

Start with simple "HelloPurr" app to introduce **Sound**, **Button**, **Screen** components and **event handling**.


Display Invisible Components in Viewer
5:09 PM
Screen1
Pet the kitty to hear it meow

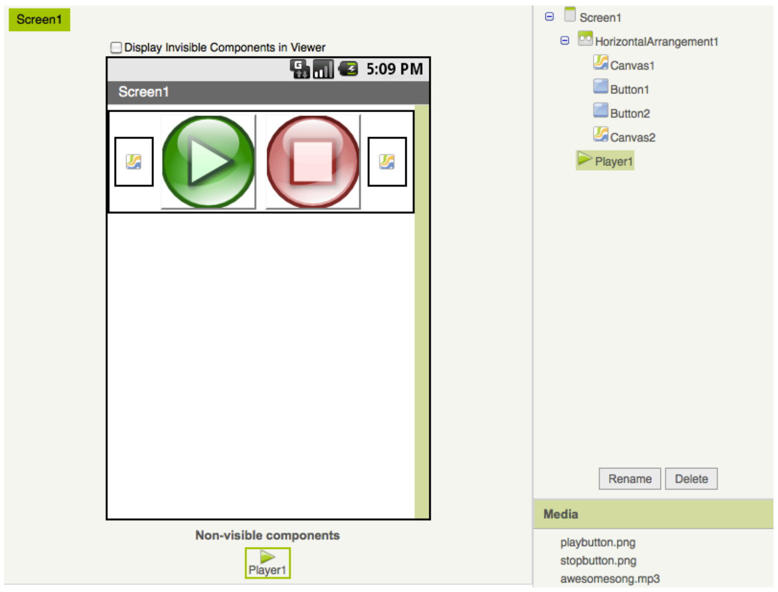


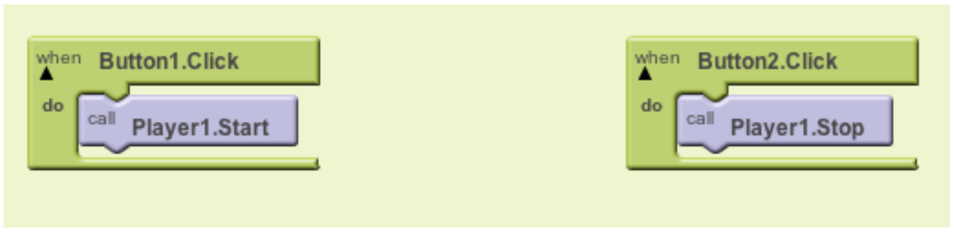
Describe the difference between **events** and **procedures** (also known as functions) in the context of HelloPurr. Button1.**Click** is a handler for an **event**, whereas Sound1.**Play** is a procedure.

Explain how to **connect to device** from the BlocksEditor, so that students can test their HelloPurr apps on their devices.

Students should partner up and modify their HelloPurr app to turn it into a Media Player with play and stop buttons. Also, introduce the **Player** component here, which can play long songs, whereas the Sound component from the HelloPurr app could only play short sounds.

Introduce the concept of layouts + **screen arrangements**. Show the students how to put the Play button and the Stop button on the same line by inserting both of them into a **HorizontalArrangement** component. Show the students how to *center* the buttons by adding empty components (such as Canvas or Label) on the left and right side of the HorizontalArrangement so that they serve as spacers.



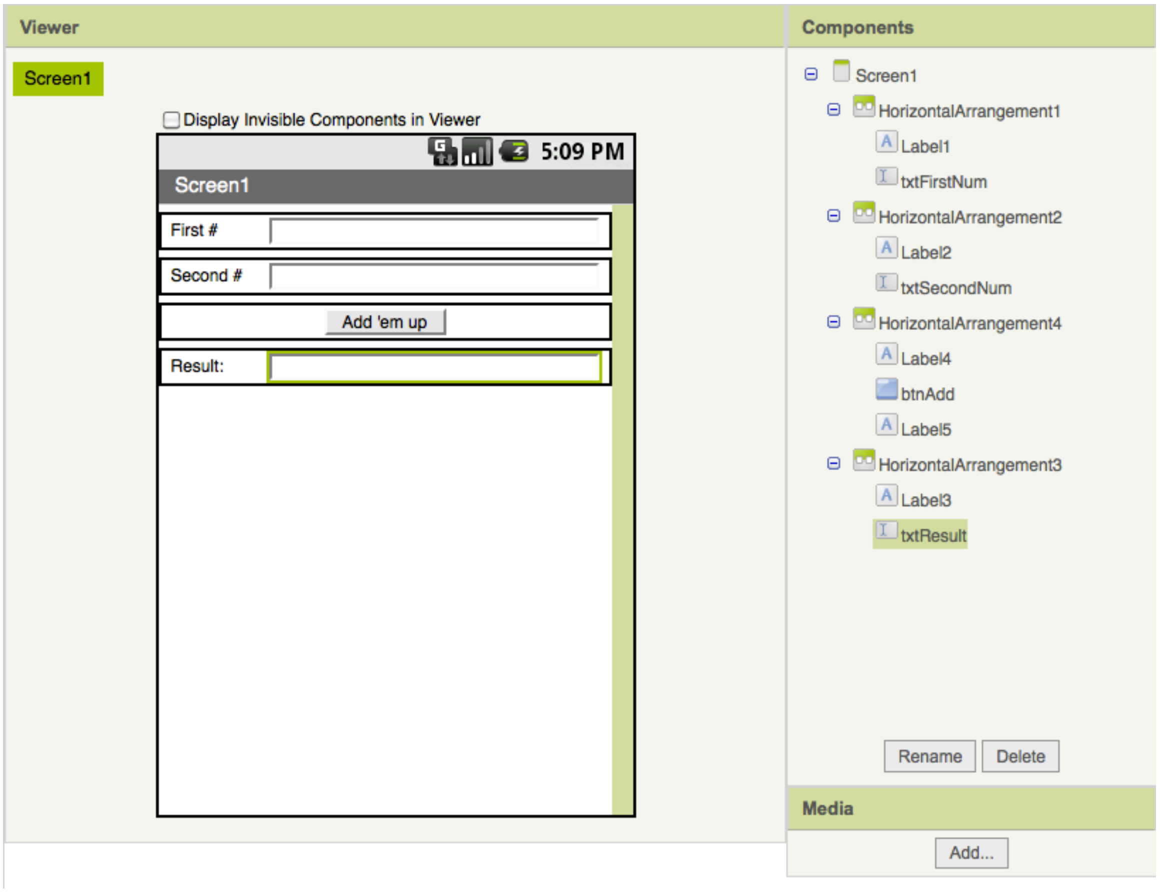


```
when Button1.Click
do   call Player1.Start

when Button2.Click
do   call Player1.Stop
```

Extra Credit: Add a pause/resume button, in addition to the play and stop buttons. Student may choose to have the 'resume' state of the pause-resume button look different from the 'pause' state -- if this is the case, teach them how to change the button's background image when the button is pressed, by setting the "Picture" property to a string representing the filename of the new button image.

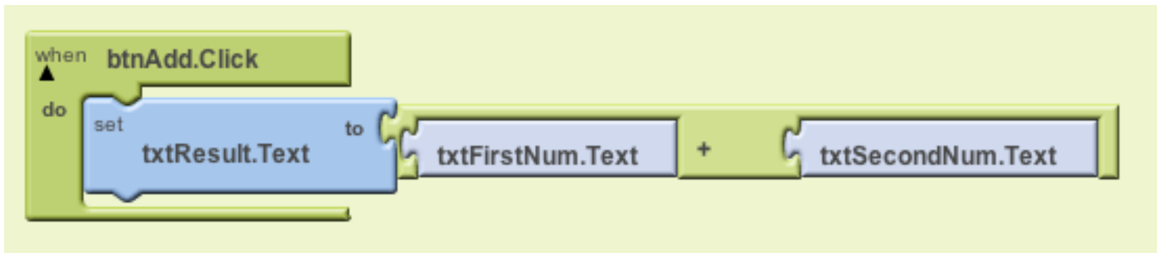
Project 2: Calculator

Introduce students to **Textbox** and **Label** components, and ask them to use these along with HorizontalArrangements and Buttons to produce the interface for a two-number adder that looks as follows.

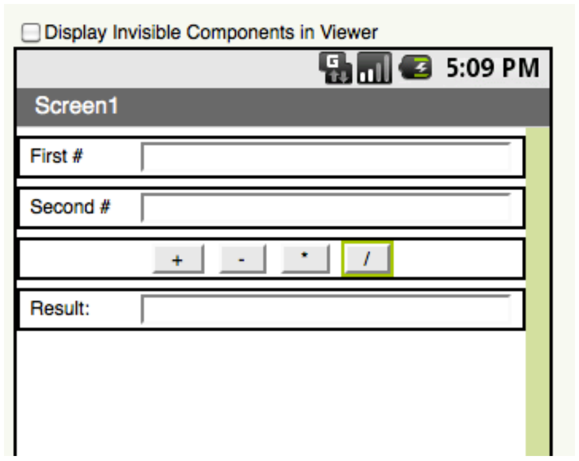


The goal of this adder is to add the two numbers typed in by the user and to indicate their sum in the result text box when the 'add' button is pressed.

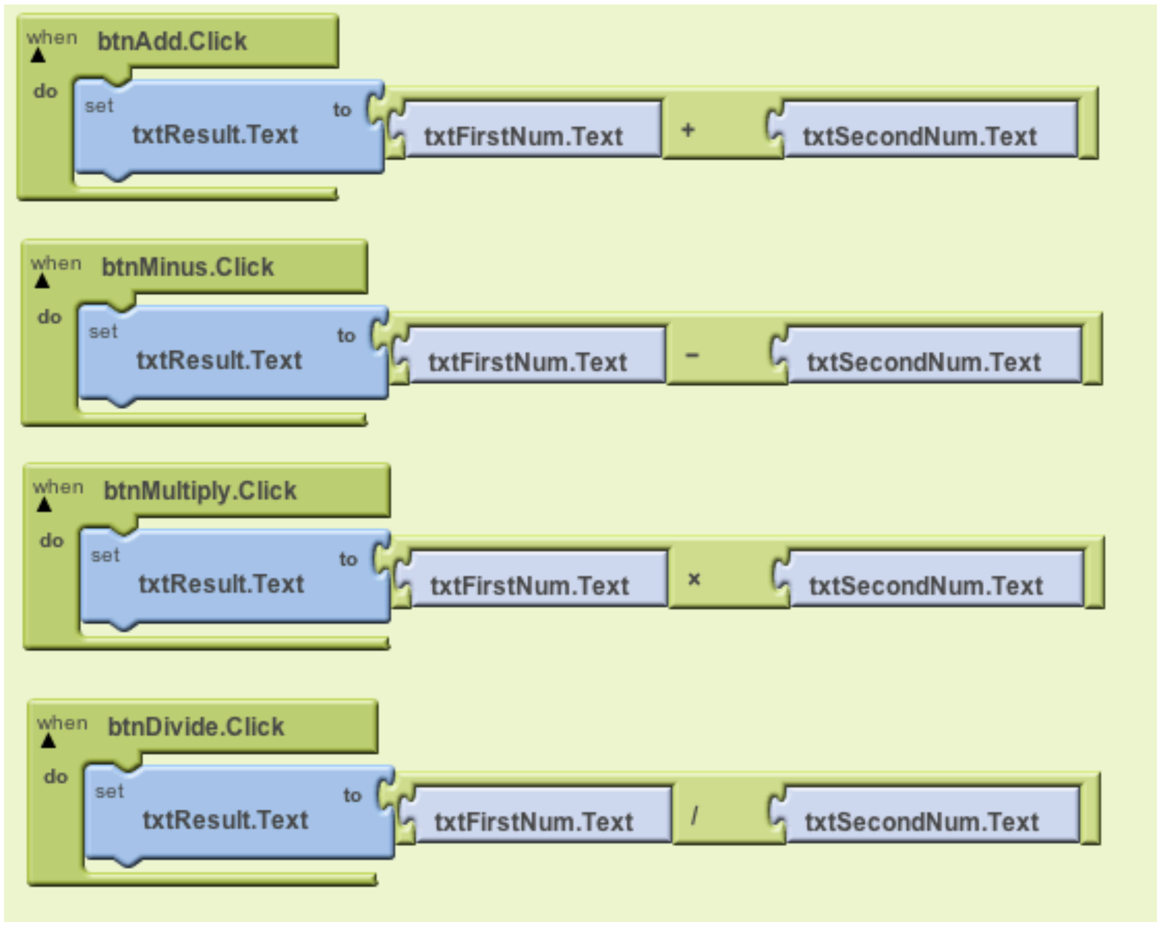
As the students implement this functionality in the Blocks Editor, they will learn about **properties** of components, and will learn to get and set their value.



Students should work individually to modify this adder and turn it into a calculator that can do all four operations on two numbers. The students should use screen arrangements to make this look nice, see below for a screenshot:

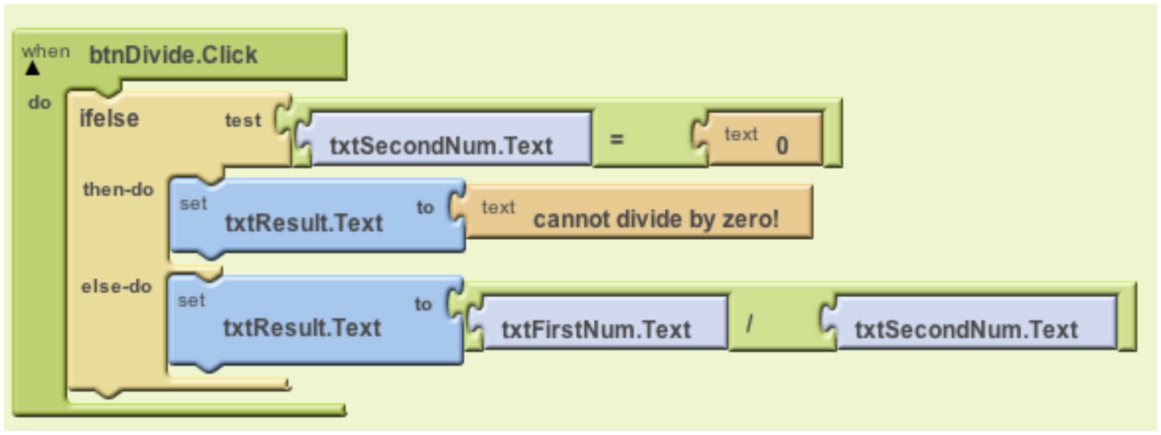


After the students have implemented the interface, ask them to follow the model of the adder they previously constructed to implement the subtraction, multiplication, and division operations.



```
when btnAdd.Click
do  set txtResult.Text to  txtFirstNum.Text + txtSecondNum.Text

when btnMinus.Click
do  set txtResult.Text to  txtFirstNum.Text - txtSecondNum.Text

when btnMultiply.Click
do  set txtResult.Text to  txtFirstNum.Text × txtSecondNum.Text

when btnDivide.Click
do  set txtResult.Text to  txtFirstNum.Text / txtSecondNum.Text
```

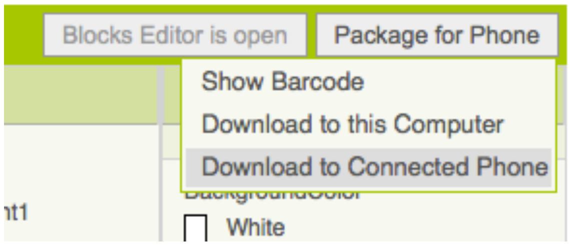
Next, teach students about basic control structures, specifically **if-else statements**. Ask students to use if statements to prevent division by zero. Here is one way of doing this:



```
when btnDivide.Click
do  ifelse  test  txtSecondNum.Text = text 0
            then-do  set txtResult.Text to  text cannot divide by zero!
            else-do  set txtResult.Text to  txtFirstNum.Text / txtSecondNum.Text
```

Extra Credit: Ask students that finish early to style their calculator by changing background color / images of the screen, buttons, textboxes, or other components.

Once all students are caught up, teach how to **package** appinventor apps to the phone so that the app can run on the phone without requiring the phone to remain connected to the computer. Ask students to package their calculators and their MediaPlayer apps and to download these to their Android devices.

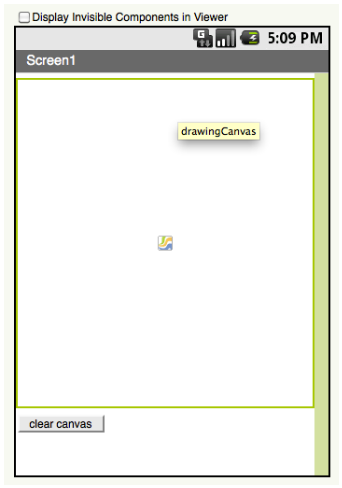


Day 2:

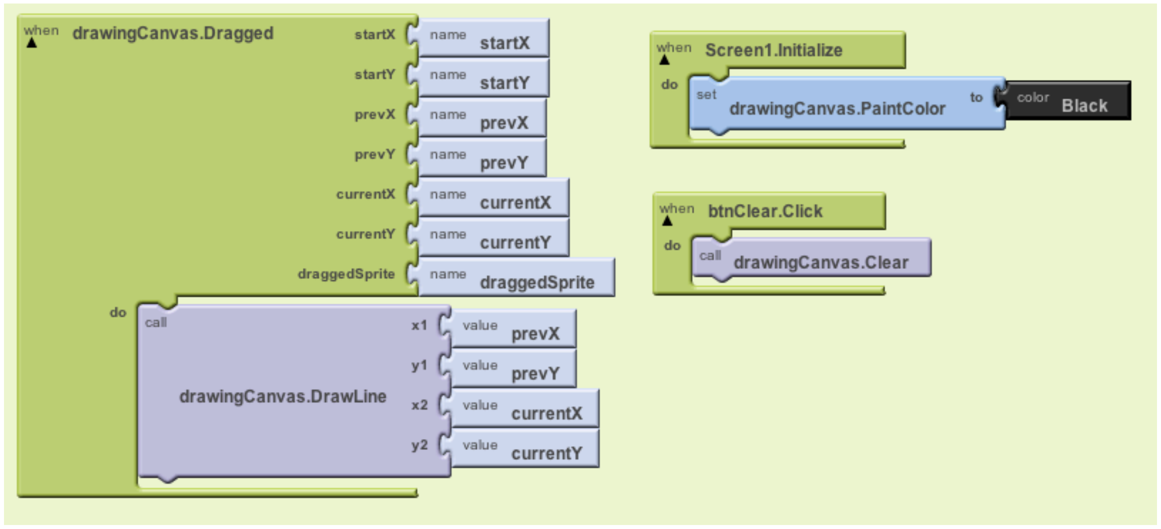
Project 3: Drawing Program

Introduce **Canvas** component as the fundamental container component for graphics (drawings, images, sprites, etc.).

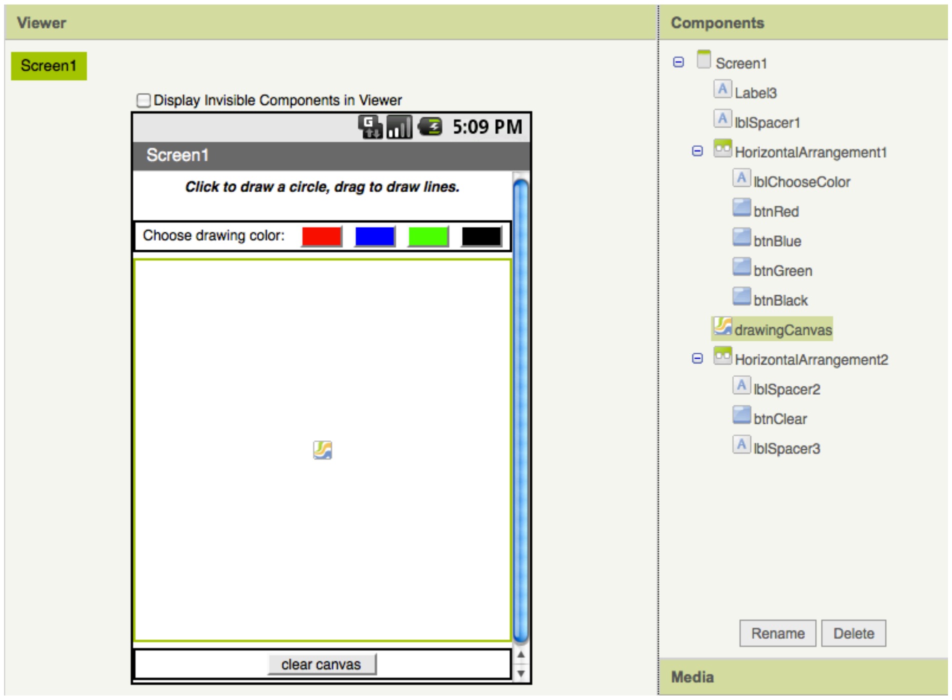
Build a simple drawing application where user drags on the canvas to draw lines and can clear canvas by clicking the clear button.

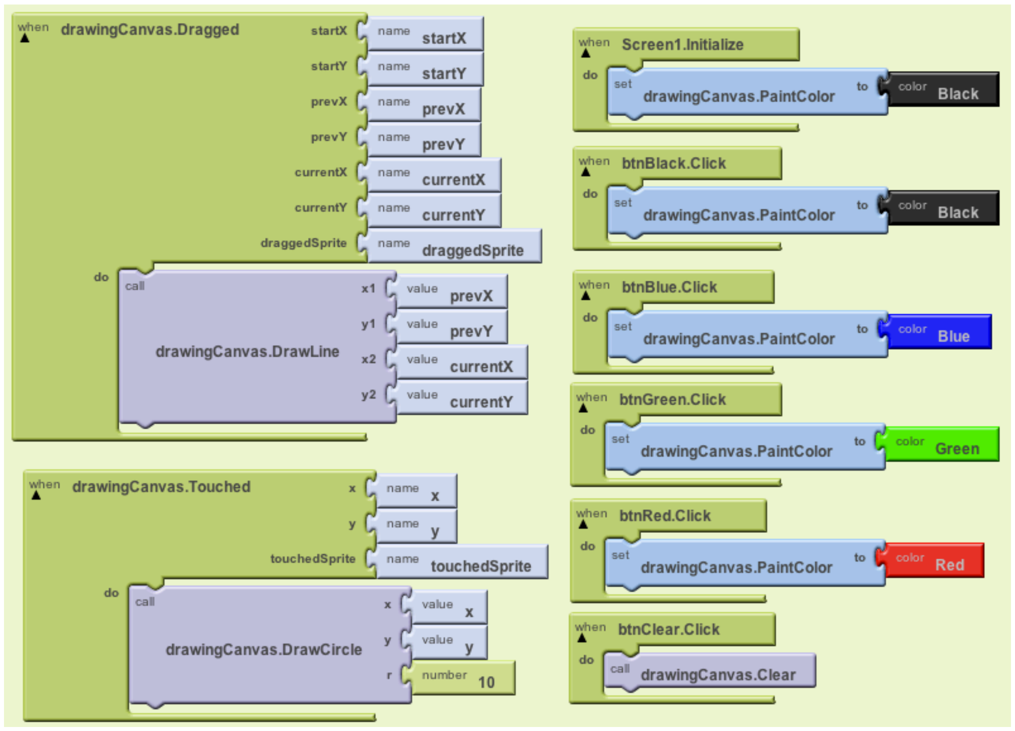


Show how to set the drawing **color** of canvas when screen (application) gets initialized, and how to handle **dragged** events.



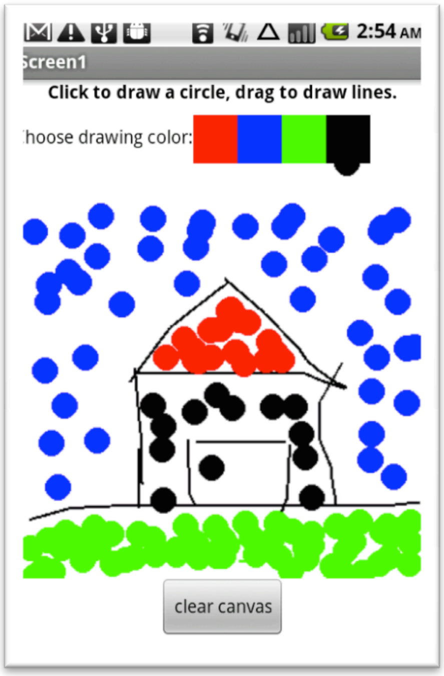
Ask students to modify the drawing program to include buttons to set the drawing color of the application, and to draw circles in response to **touched** events.





```
when drawingCanvas.Dragged
    startX        name startX
    startY        name startY
    prevX         name prevX
    prevY         name prevY
    currentX      name currentX
    currentY      name currentY
    draggedSprite name draggedSprite
do  call drawingCanvas.DrawLine
        x1 value prevX
        y1 value prevY
        x2 value currentX
        y2 value currentY

when drawingCanvas.Touched
    x             name x
    y             name y
    touchedSprite name touchedSprite
do  call drawingCanvas.DrawCircle
        x value x
        y value y
        r number 10

when Screen1.Initialize
do  set drawingCanvas.PaintColor to color Black

when btnBlack.Click
do  set drawingCanvas.PaintColor to color Black

when btnBlue.Click
do  set drawingCanvas.PaintColor to color Blue

when btnGreen.Click
do  set drawingCanvas.PaintColor to color Green

when btnRed.Click
do  set drawingCanvas.PaintColor to color Red

when btnClear.Click
do  call drawingCanvas.Clear
```

If students finish early, ask them to make a beautiful picture using your drawing app, like this one:



The student with the best drawing wins a chocolate :)

Project 4: Balls, Sprites, and Motion of Objects

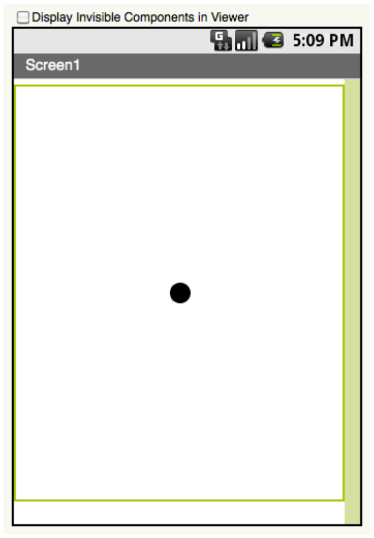
Introduce the **Ball** component as one of two Sprite components in AppInventor that have the ability to move around on a canvas. The second sprite component is **ImageSprite**, which is similar to a ball in that it can be moved around on a canvas, but it is visually represented by a custom Image that is uploaded by the AppInventor developer.

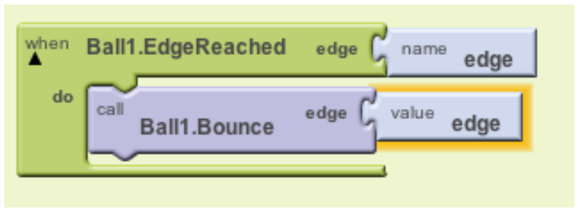
There are many ways to make Balls and ImageSprites move, but in this course we'll discuss the four of them (listed below) that are most applicable to game development:

1) Automatically (using built-in properties)
2) Using buttons
3) Using the AppInventor Clock component
4) Using Accelerometer

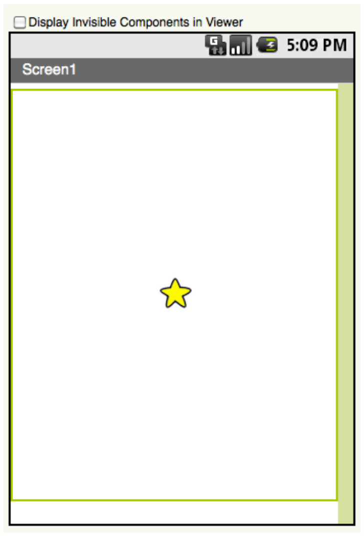
Motion Method 1: Automatically

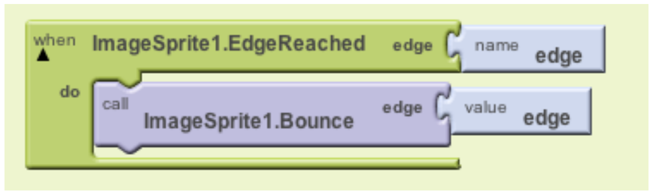
Introduce sprite properties such as **Heading**, **Speed**, **X**, **Y**, and **Interval** that are required for automated motion of the sprite, and show students how to set these properties from the Properties panel in the Designer view. Create a simple app where the ball bounces around the screen, using the 'EdgeDetected' event handler and the 'Bounce' procedure in the blocks editor.





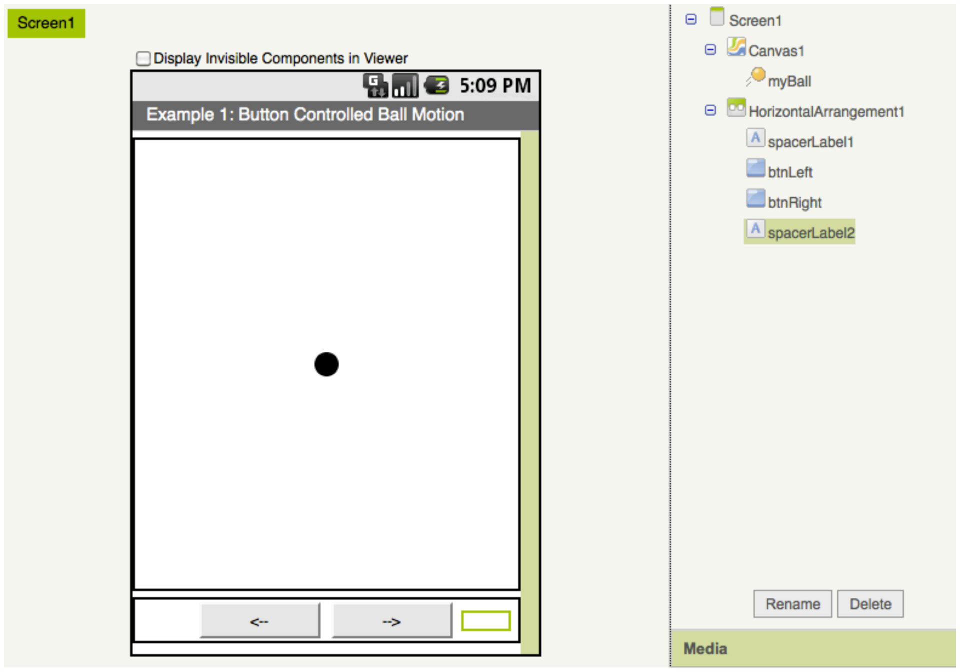
Then, ask the students (individually) to replace the ball with an **ImageSprite**, using an image of their choice (e.g. a fancy ball image from Google), but make it do the same sort of bouncing around using 'edge detected'.





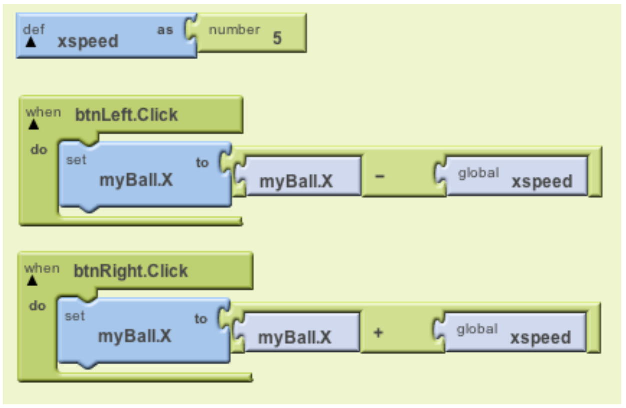
Motion Method 2: Button-controlled ball motion:

Ask students to create an AppInventor project with a single ball on a canvas and left/right buttons below the canvas, as shown below:



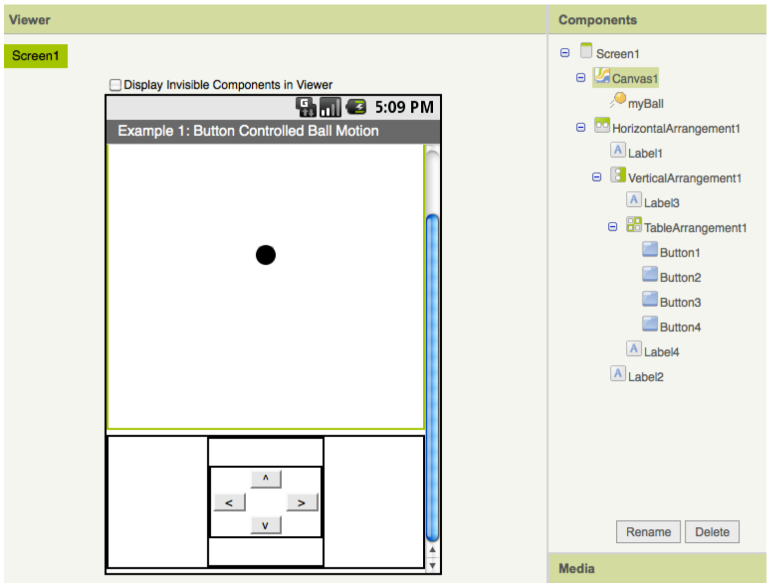
In order to make the ball move, we will update the ball's x position as the left and right buttons are pressed. The amount of pixels that we move the ball *each* time the buttons are clicked is known as the ball's *speed* (in the x direction).

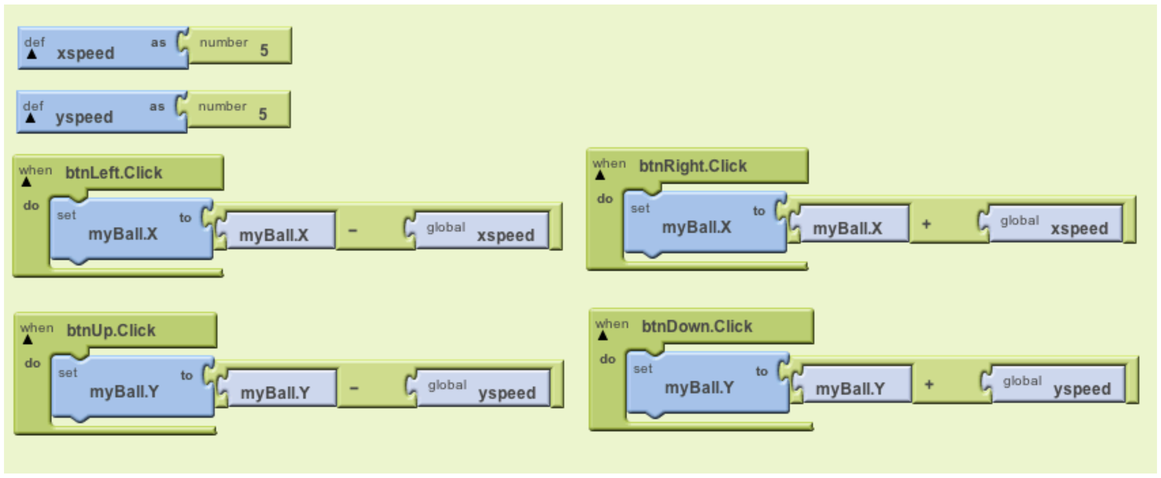
Introduce the concept of **variables**, and discuss how to define your own, and how to access and set their value. Then, ask students to create a variable representing the speed of the ball in the x direction. The blocks code for making the ball move then looks as follows:



```
def xspeed as number 5

when btnLeft.Click
do set myBall.X to myBall.X - global xspeed

when btnRight.Click
do set myBall.X to myBall.X + global xspeed
```

Ask students to *extend* this app for up and down motion controlled by up and down arrows:

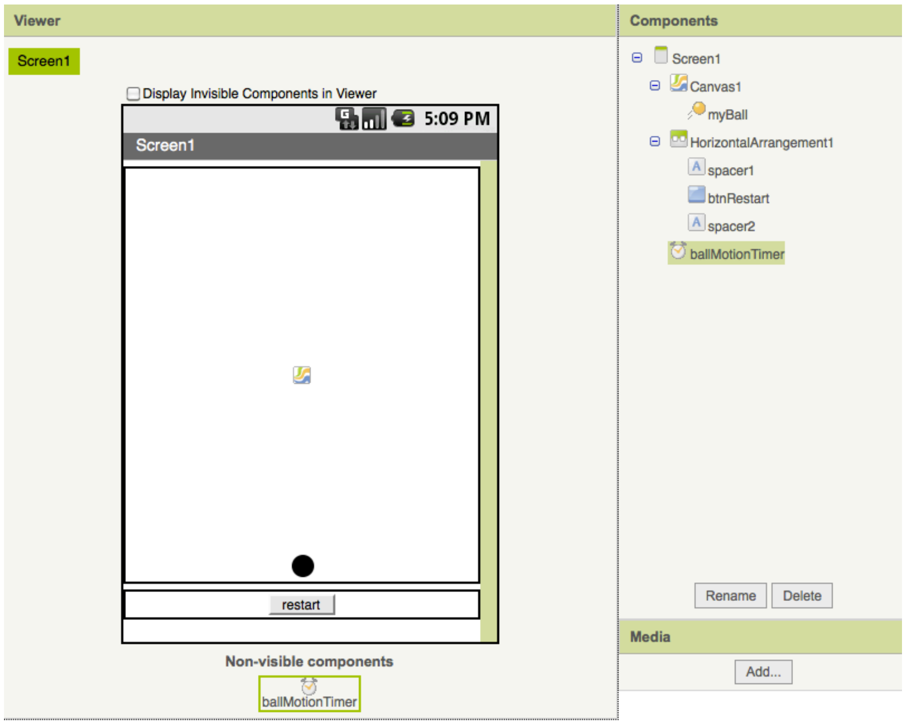



def xspeed as number 5
def yspeed as number 5
when btnLeft.Click do set myBall.X to myBall.X - global xspeed
when btnRight.Click do set myBall.X to myBall.X + global xspeed
when btnUp.Click do set myBall.Y to myBall.Y - global yspeed
when btnDown.Click do set myBall.Y to myBall.Y + global yspeed

Motion Method 3: Using the AppInventor Clock component

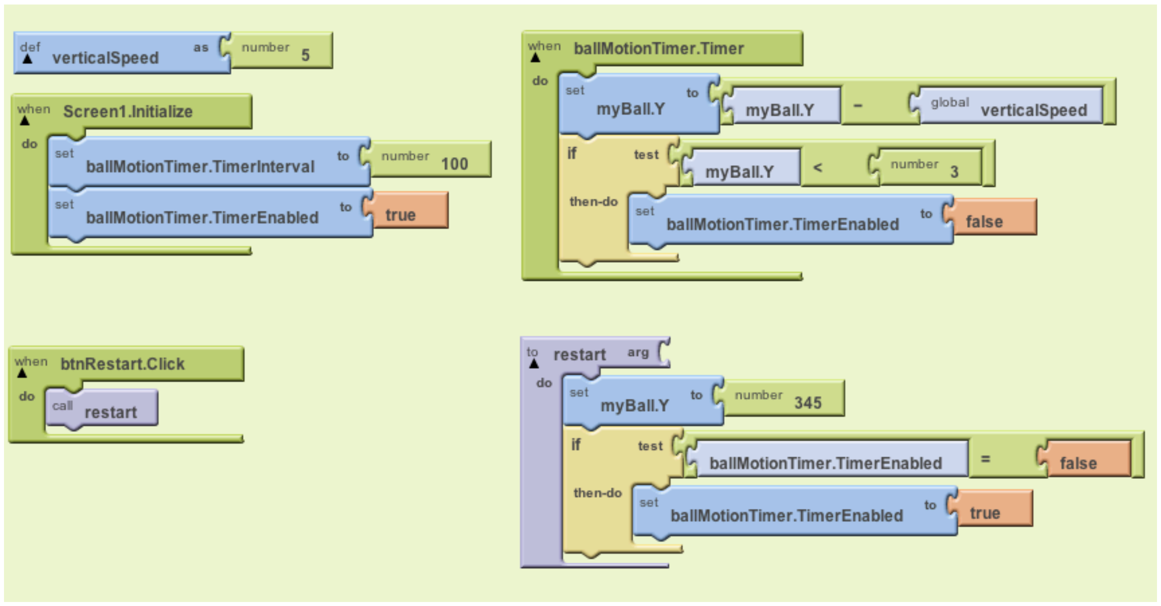
Introduce the **Clock** component, which is a timer that fires every few milliseconds (as specified by the **TimerInterval** property, which can be set either in the designer view or in the blocks editor).

Ask students to create a new project with a Ball, Canvas, and Clock (which will show up below the screen as a **non-visible component**).



We now make the ball move up a few pixels (as determined by the value of the speed variable) every time the timer fires. Once the ball reaches the top of the screen, we ask the timer to stop firing, and we wait for the user to click the 'restart' button. Clicking the restart button causes the ball to reset its location to the bottom of the screen and turns on the timer once again.

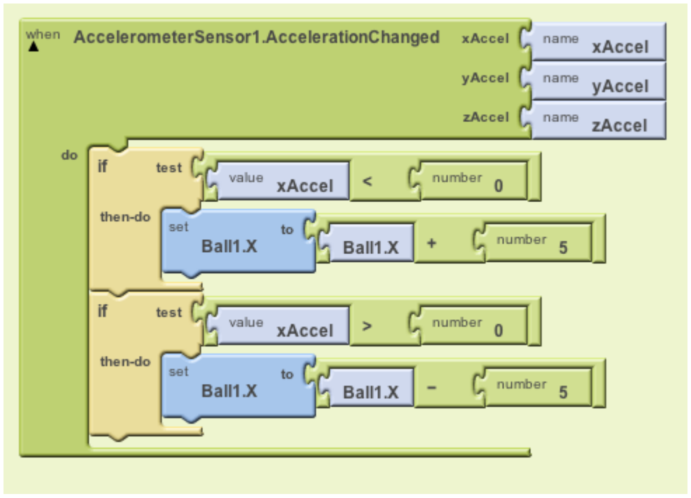
We use this opportunity to teach how to define and call your own **procedure**, by turning the 'restart' action into its own procedure. The blocks code is shown below:



```
def verticalSpeed as number 5

when Screen1.Initialize
do  set ballMotionTimer.TimerInterval to number 100
    set ballMotionTimer.TimerEnabled to true

when btnRestart.Click
do  call restart

when ballMotionTimer.Timer
do  set myBall.Y to myBall.Y - global verticalSpeed
    if test myBall.Y < number 3
    then-do set ballMotionTimer.TimerEnabled to false

to restart arg
do  set myBall.Y to number 345
    if test ballMotionTimer.TimerEnabled = false
    then-do set ballMotionTimer.TimerEnabled to true
```

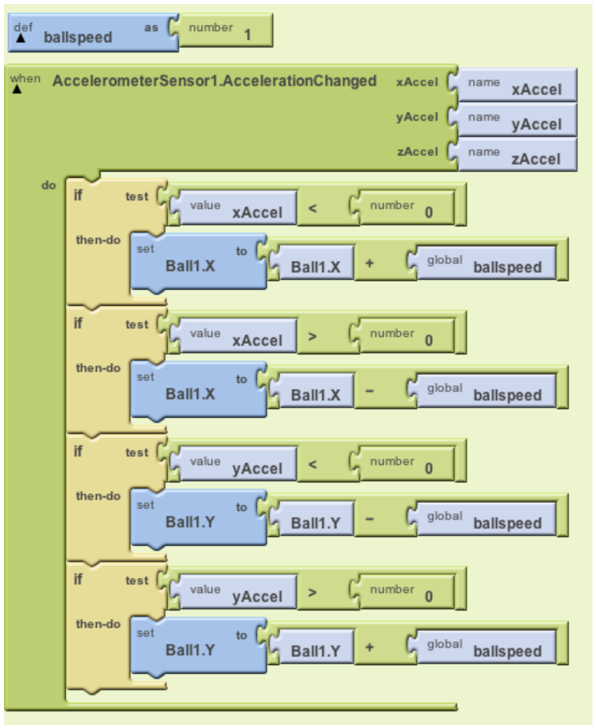
Motion Method 4: Using the Accelerometer

The Accelerometer is a built-in sensor in Android devices that senses changes in acceleration. Here, we use the accelerometer to make the ball move left and right as the phone is tilted left and right, hence avoiding the need for buttons and other forms of tactile input.



```
when AccelerometerSensor1.AccelerationChanged  xAccel name xAccel
                                               yAccel name yAccel
                                               zAccel name zAccel
do  if test value xAccel < number 0
       then-do set Ball1.X to Ball1.X + number 5
    if test value xAccel > number 0
       then-do set Ball1.X to Ball1.X - number 5
```

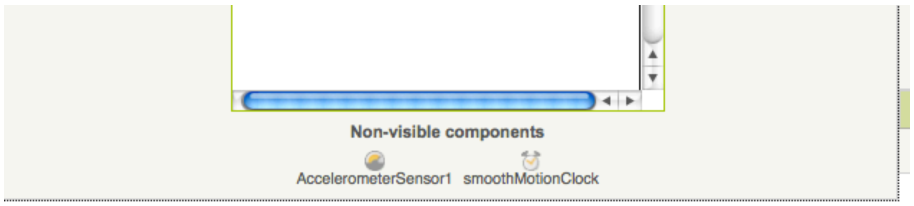
If students finish early, ask them to extend the program to also detect acceleration changes in the y direction (to move the ball up and down):



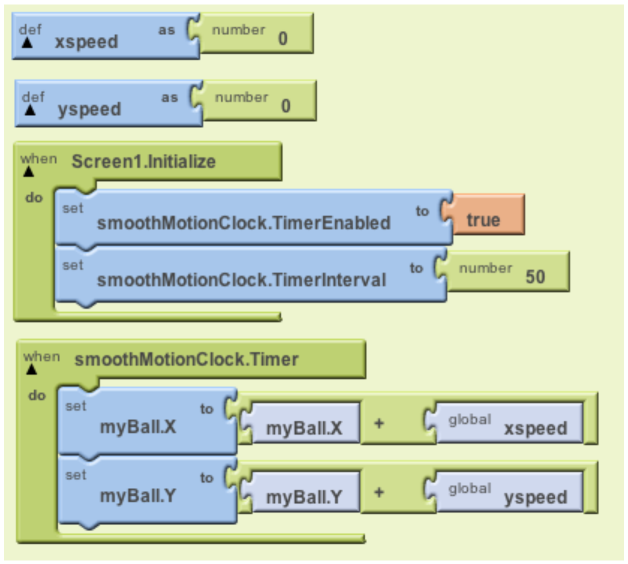
```
def ballspeed as number 1
when AccelerometerSensor1.AccelerationChanged  xAccel name xAccel
                                               yAccel name yAccel
                                               zAccel name zAccel
do  if test value xAccel < number 0
       then-do set Ball1.X to Ball1.X + global ballspeed
    if test value xAccel > number 0
       then-do set Ball1.X to Ball1.X - global ballspeed
    if test value yAccel < number 0
       then-do set Ball1.Y to Ball1.Y - global ballspeed
    if test value yAccel > number 0
       then-do set Ball1.Y to Ball1.Y + global ballspeed
```

Extra Credit: Students will note that while the above code works, the motion of the ball is not smooth. In order to achieve smooth motion, students should use the **Clock** component *in conjunction* with the **Accelerometer** sensor, following the steps below.

Smooth Motion Step 1: Add a Clock component to the project. In the below screenshots, the clock is called "smoothMotionClock".

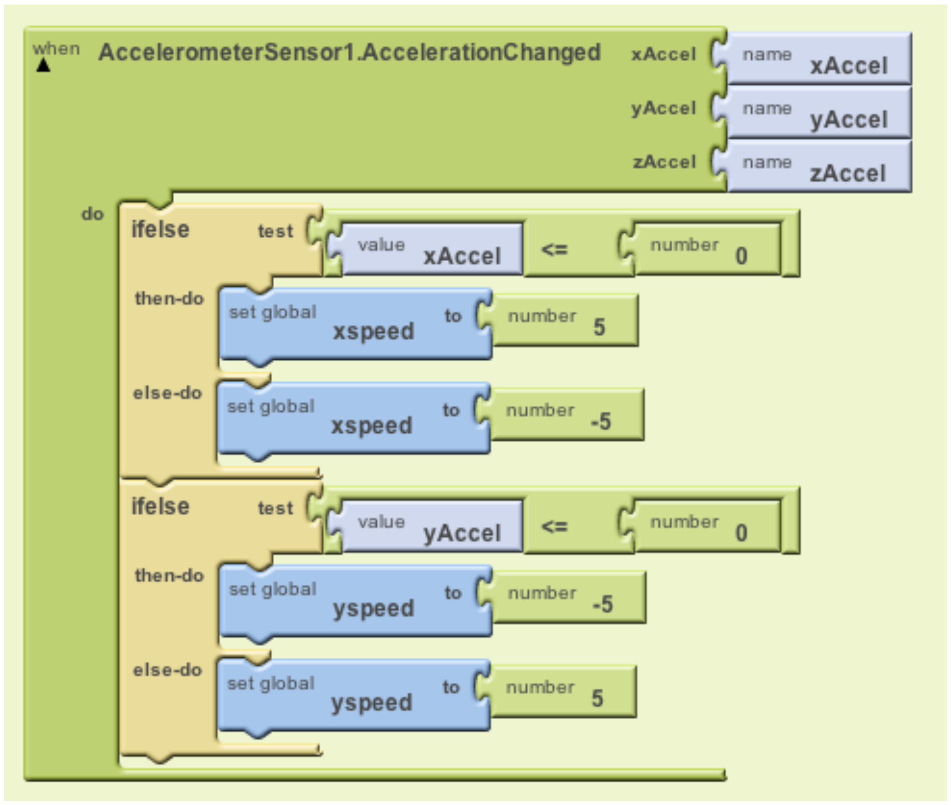


Smooth Motion Step 2: In the blocks code, create variables for speed in the x and y direction, initialize the clock's timer interval in the Screen's Initialize event, and handle the clock's Timer event by updating the ball's x and y location by the appropriate speed variables.



```
def xspeed as number 0

def yspeed as number 0

when Screen1.Initialize
do  set smoothMotionClock.TimerEnabled to true
    set smoothMotionClock.TimerInterval to number 50

when smoothMotionClock.Timer
do  set myBall.X to myBall.X + global xspeed
    set myBall.Y to myBall.Y + global yspeed
```

Smooth Motion Step 3: Inside the AccelerometerSensor component's AccelerationChanged event handler, instead of updating the ball's position as you did in the non-smooth version of this application, simply set the value of the x and y speed variables. Now, the ball's position will be updated by the Clock timer and will be unaffected by factors such as how quickly the phone is tilted, resulting in regular and smooth motion.

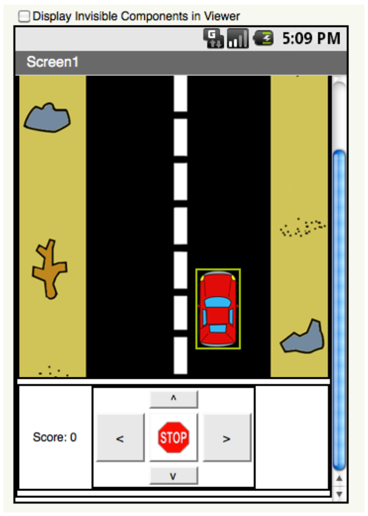


```
when AccelerometerSensor1.AccelerationChanged   xAccel  name xAccel
                                                yAccel  name yAccel
                                                zAccel  name zAccel
  do
    ifelse   test   value xAccel  <=  number 0
      then-do   set global xspeed to   number 5
      else-do   set global xspeed to   number -5
    ifelse   test   value yAccel  <=  number 0
      then-do   set global yspeed to   number -5
      else-do   set global yspeed to   number 5
```

Day 3:

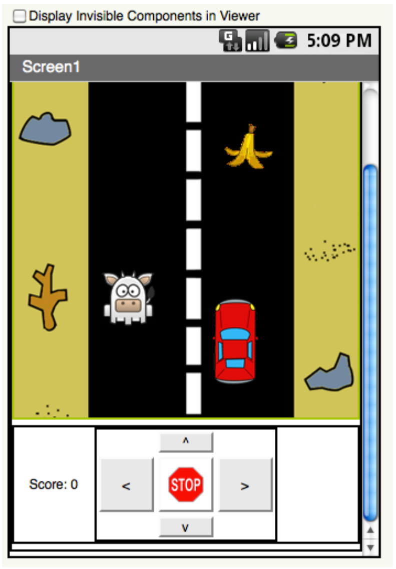
Project 5: Car on a Road

Review previous day's material -- Canvas, Drawing, and motion of sprites -- by building a simple game where a car (represented by an ImageSprite) moves left and right on a road (a Canvas with an image as a background):

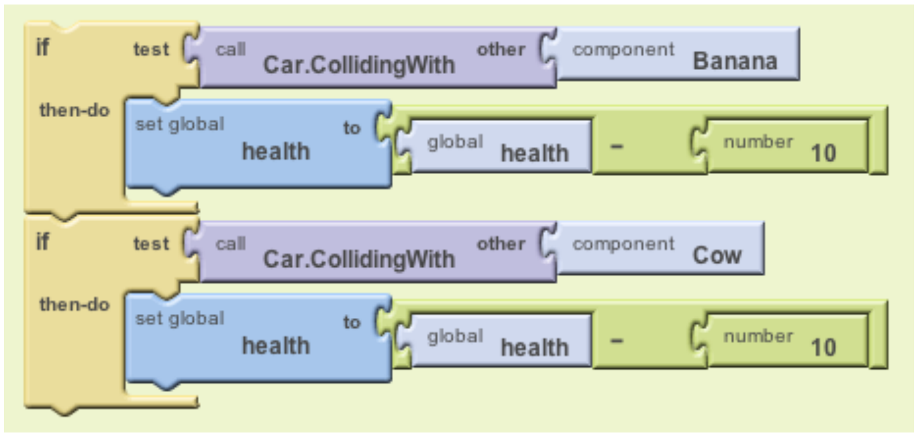


The students may implement the motion of the car using either the Clock component, buttons, or the accelerometer.

Using this same project as a starting point, introduce students to **collision detection**, a very useful facet of game development. Ask them to add one or more obstacles (ImageSprite objects) onto the road, e.g. the banana and the cow in the following image:

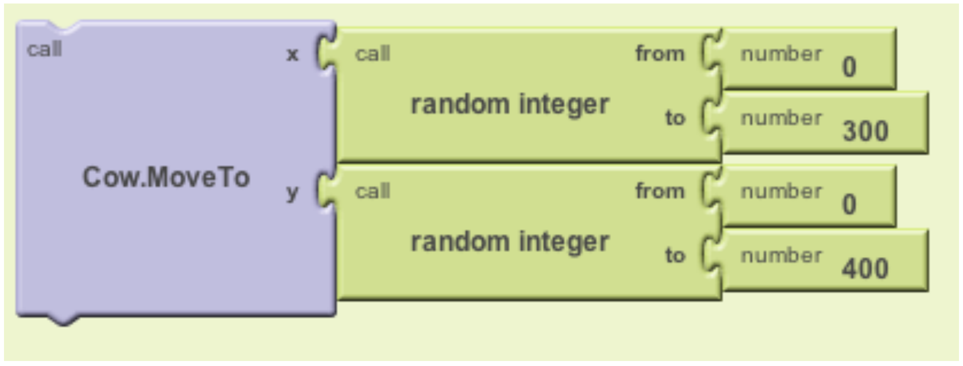


In the blocks code, ask students to insert if-statements (right after the code that updates the car's position) to check if the car is colliding with other obstacles:


```
if       test    call Car.CollidingWith   other   component Banana
then-do  set global health to   global health  -  number 10
if       test    call Car.CollidingWith   other   component Cow
then-do  set global health to   global health  -  number 10
```

Next, introduce students to **random-number generation** by showing them how to make the obstacles on the road appear in random (x,y) coordinate locations using the *random integer* procedure to pick random values for x and y.

In the below blocks code, the Cow's x value is picked randomly from within the range [0, 300] and the y value is picked randomly from the range [0, 400]:

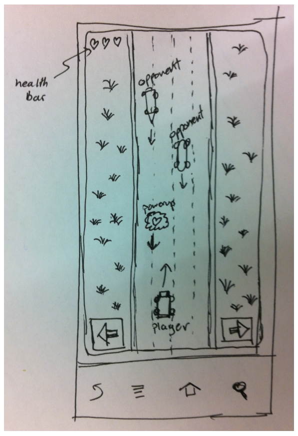

```
call Cow.MoveTo
  x   call random integer   from   number 0
                            to     number 300
  y   call random integer   from   number 0
                            to     number 400
```

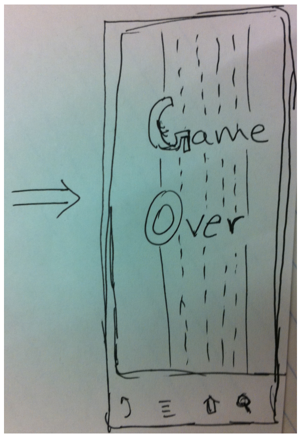
Extra Credit: If students finish early, have them attempt to complete this car obstacle-avoidance game by animating the obstacles such that they appear at the top of the screen, move from top to bottom, and then reset to the top of the screen (at a random x location) and come down again. The students can accomplish this either manually using Clock components or automatically using the built-in ImageSprite properties of Interval, Speed, and Heading (the heading would be 270 degrees if the objects are moving straight down). Note: If multiple obstacles are to be moved simultaneously but at different rates, one Clock component might be necessary for each obstacle.

Project 6: Paper Prototyping

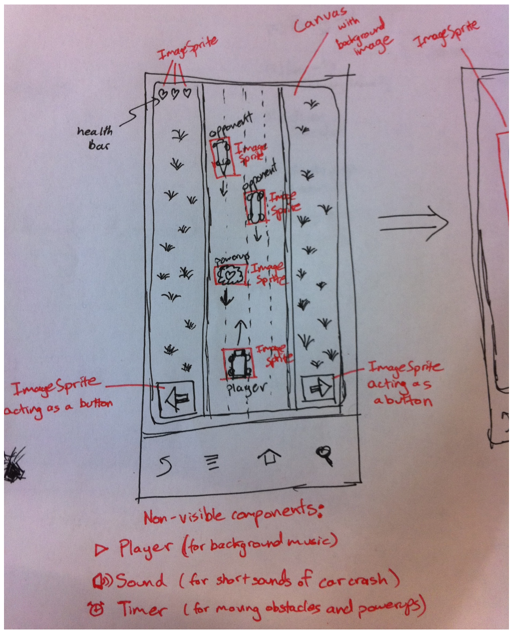
Ask students to brainstorm ideas for games they wish to create by the end of the course (they should be shown examples of existing AppInventor / Android games in order to get their creative juices flowing). Then take them through the steps below.

Step 1: Draw, with pencil or pen, a prototype of the game you wish to create. If the game will have multiple screens, draw out each of those screens.

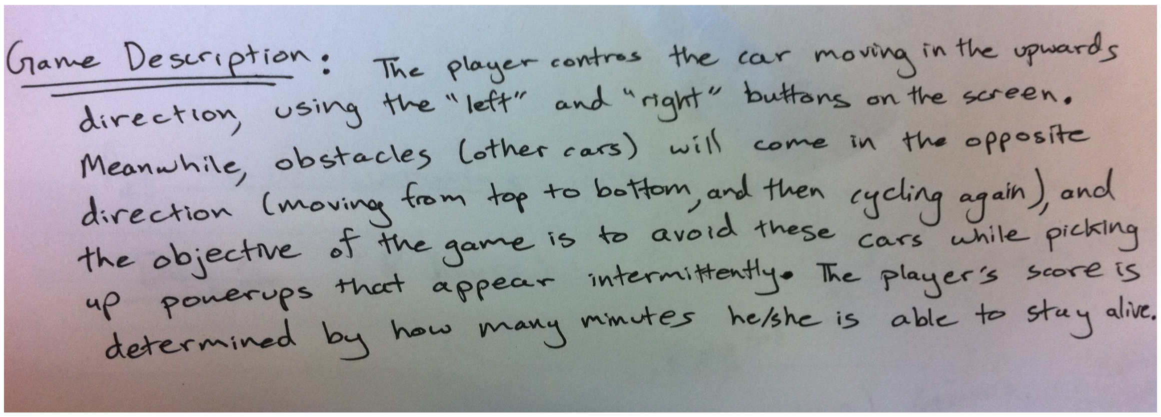




Step 2: On top of the drawings for each screen, write down the names of the AppInventor components you wish to use for each object in the game.


ImageSprite
Canvas with background image
ImageSprite
health bar
opponent
Image Sprite
opponent
Image Sprite
powerup
Image Sprite
Image Sprite
Player
ImageSprite acting as a button
ImageSprite acting as a button
Non-visible components:
Player (for background music)
Sound (for short sounds of car crash)
Timer (for moving obstacles and powerups)

Step 3: Write a brief description of the game on a piece of paper. Make the description read-able and easily understandable, you will be sharing it with your peers.


Game Description: The player contros the car moving in the upwards direction, using the "left" and "right" buttons on the screen. Meanwhile, obstacles (other cars) will come in the opposite direction (moving from top to bottom, and then cycling again), and the objective of the game is to avoid these cars while picking up powerups that appear intermittently. The player's score is determined by how many minutes he/she is able to stay alive.

Step 4: Prototype-sharing and Feedback session. Each group or individual should leave their paper prototypes and written description of the game on their table and rotate one spot to the left, where the group can evaluate the neighbor's game. The group should write down any feedback or suggestions they have on the piece of paper that contains the Game Prototype / Description, and then all groups should rotate another spot to the left. This continues until each group has rotated through all tables and provided feedback to every other group.

For the rest of the day, students should work on their games in pairs or individually. The Instructor should help groups on a 1-on-1 basis, teaching them topics that they require for their games but haven't yet learned.

Day 4:

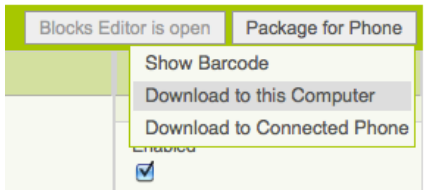
Students should work on their games as individuals or pairs. Instructor should assist groups on a 1-on-1 basis, providing help where necessary and teaching new topics that haven't yet been covered to the different groups.

Day 5:

Because this is the final day of the course, students will (hopefully) be extremely excited, energetic, and eager to work on their final projects. If there are any specific topics that they should learn before the end of the course, the morning would be the right time since by the evening the students will be busy making last-minute changes to their games.

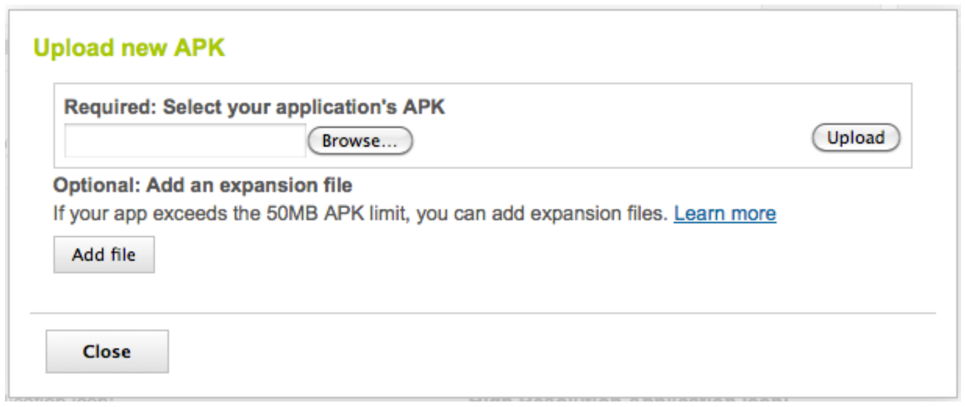
So, before asking the students to begin working on their projects for the day, show them how to **publish apps to Google Play** (formerly known as Android Market) so that the general public can download their apps from the internet to their phone. The steps are outlined below.

Step 1: Click "Download to this Computer" from the designer view of the AppInventor site in order to generate an .apk file for your AppInventor project.



Step 2: Then go to https://play.google.com/apps/publish/Home. If you are not registered as a Google Play developer, you will need to pay a registration fee, create a developer profile, and accept some agreements. After you've done that (or if you're already a registered developer) you should see the Google Play Developer Console. From this console, click the "Upload Application" button to being the process.

Step 3: Upload your .apk file when prompted.

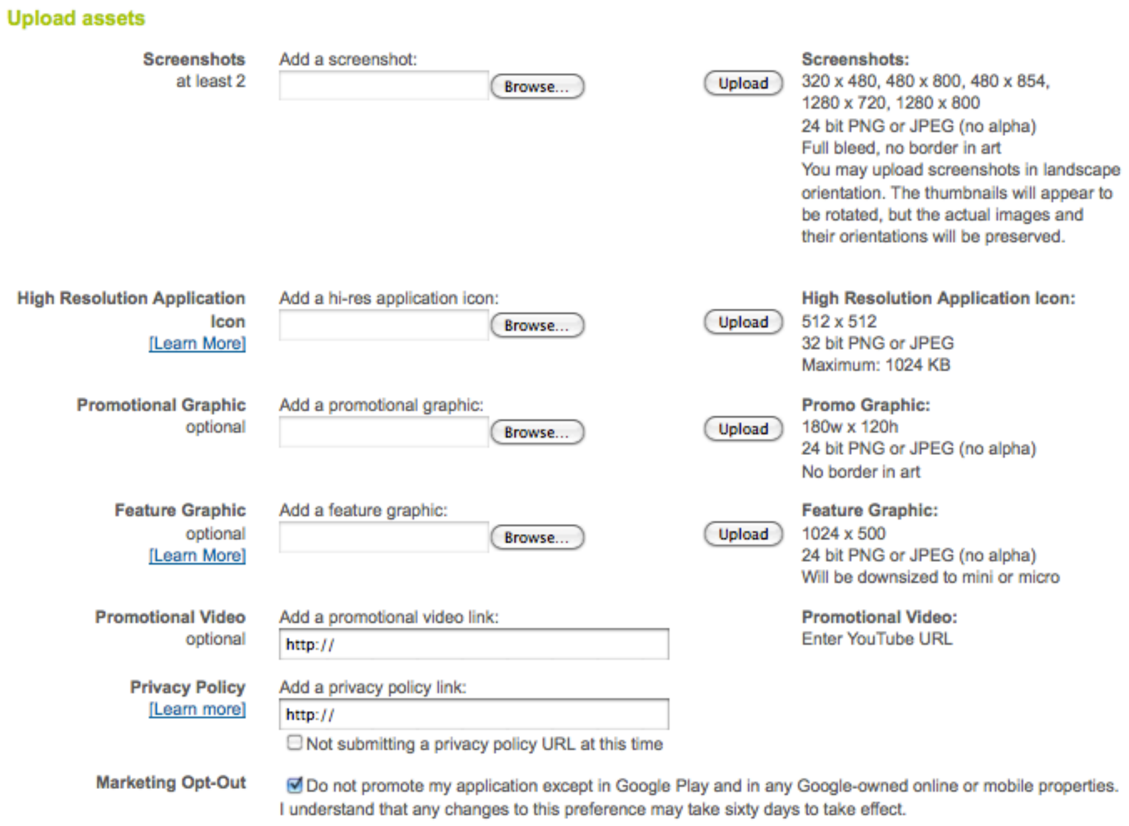

Upload new APK

Required: Select your application's APK

Optional: Add an expansion file

If your app exceeds the 50MB APK limit, you can add expansion files. Learn more

Step 4: Provide other necessary information, e.g. the image icon to be used for your Android application, and the screenshots that should show up when an Android user tries to download your application from Google Play.


Upload assets

Screenshots at least 2	Add a screenshot:	**Screenshots:** 320 x 480, 480 x 800, 480 x 854, 1280 x 720, 1280 x 800; 24 bit PNG or JPEG (no alpha); Full bleed, no border in art; You may upload screenshots in landscape orientation. The thumbnails will appear to be rotated, but the actual images and their orientations will be preserved.
High Resolution Application Icon [Learn More]	Add a hi-res application icon:	**High Resolution Application Icon:** 512 x 512; 32 bit PNG or JPEG; Maximum: 1024 KB
Promotional Graphic optional	Add a promotional graphic:	**Promo Graphic:** 180w x 120h; 24 bit PNG or JPEG (no alpha); No border in art
Feature Graphic optional [Learn More]	Add a feature graphic:	**Feature Graphic:** 1024 x 500; 24 bit PNG or JPEG (no alpha); Will be downsized to mini or micro
Promotional Video optional	Add a promotional video link: http://	**Promotional Video:** Enter YouTube URL
Privacy Policy [Learn more]	Add a privacy policy link: http:// ☐ Not submitting a privacy policy URL at this time	
Marketing Opt-Out	☑ Do not promote my application except in Google Play and in any Google-owned online or mobile properties. I understand that any changes to this preference may take sixty days to take effect.	

Step 5: Click "Publish".

For the rest of the day, the students should work on finishing the games they had started developing the previous day, and the instructor should work with them on a one-on-one basis.

During the final hour of the course, the students should present their apps to each other.

Chapter 3

Teaching Experiences and Lessons Learned

Teaching in Bangalore, India.

The author taught a 1-week (40 hours split over 5 days) course on Android Game Development using AppInventor to a group of 10 high-school students in Bangalore during December 2011. The students were sharp but had no prior programming experience. Yet when exposed to the curriculum outlined in Chapter 2, these students were able to grasp AppInventor's interface quickly and by the end of the week, each of them had built his very own game from scratch.

During this workshop, the author worked with the students to complete certain exercises together (with the entire class) until Day 3, but from that point on, he allowed each student to work on his own app. Some photos of the students are included below:

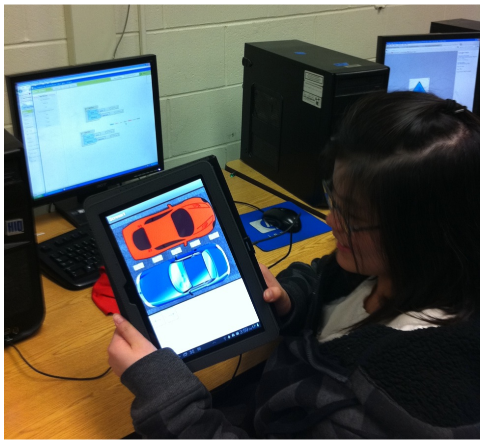
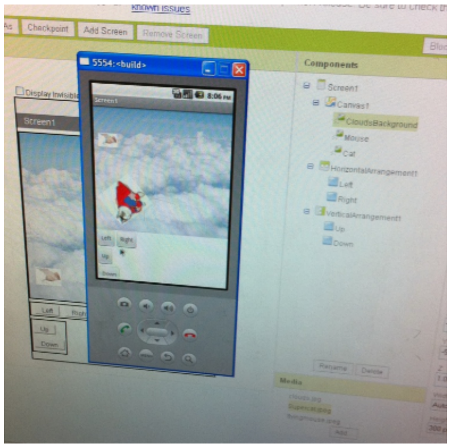
Students, along with the author (their instructor) during a field-trip to Google's office in Bangalore.

Students showing off the games they developed using AppInventor.

Teaching in Boston.

The author taught a series of 3 afterschool workshops (2-hours each) on 3 different weeks at the John D. O'Bryant School of Math and Science in Boston. The focus was Game Development using AppInventor, but because the total length of instruction time was 6 hours, the author chose not to follow his 40-hour curriculum and instead chose a more improvisational approach that was dependent entirely on the speed at which students picked up new concepts.

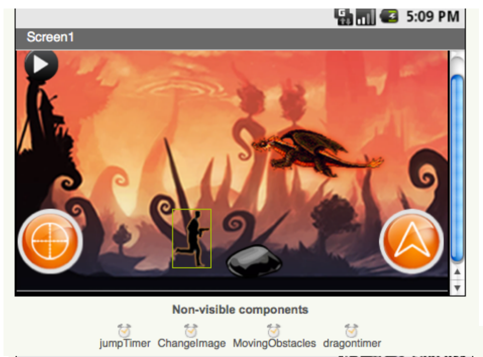
Because each workshop was scheduled for a different week, there was very little continuity between the 3 workshops and during the 2nd and 3rd workshops, students had completely forgotten what they had learned during the previous one. As a result, the students in the course did not get nearly as far as the students in India had gotten on their projects, but they still managed to make some simple games (e.g. a user-controlled cat chasing a mouse that is bouncing around the screen, or a car game that has two cars moving in opposite directions on a road). Pictures from this experience are included below:

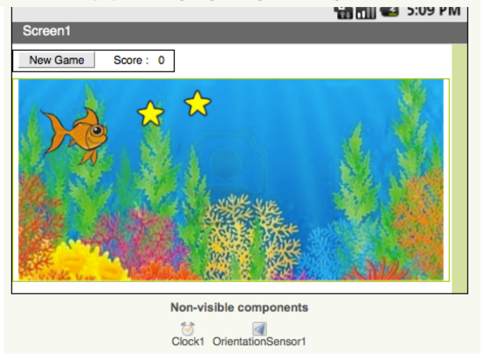



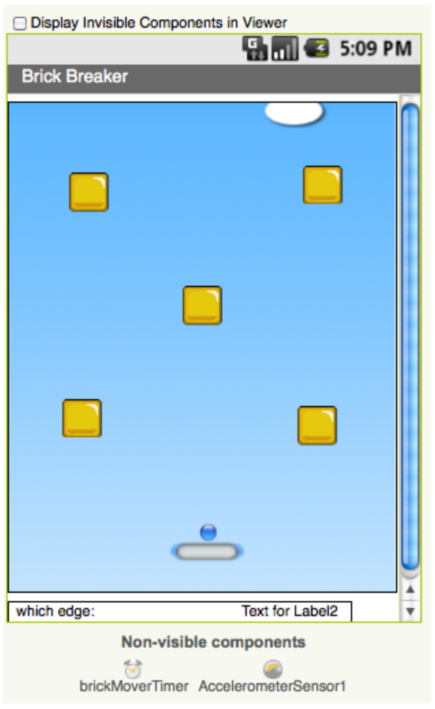
Left: a car-racing game for Android tablets. Right: a game where the cat chases the mouse. Bottom: happy students from the course.

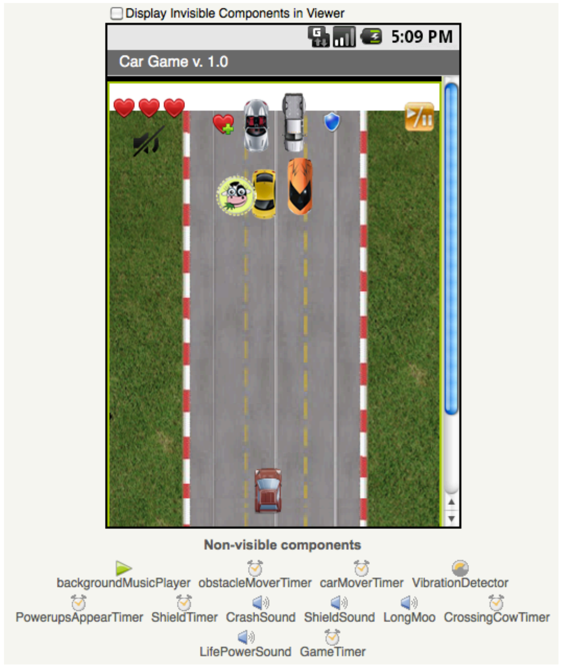
Pictures of student projects.

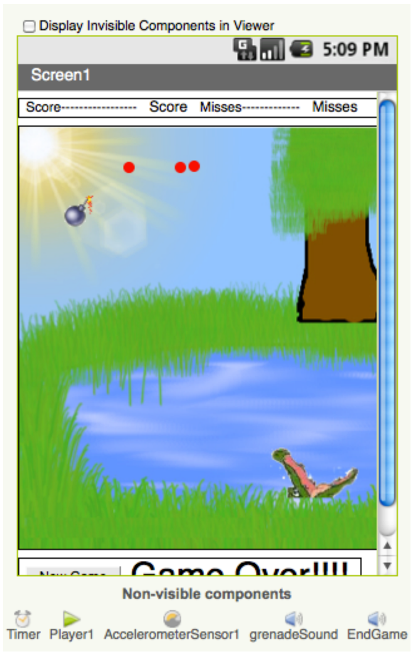
Included below are screenshots of some of the nicer games developed by the author's students in India and Boston.






Display Invisible Components in Viewer
5:09 PM
Brick Breaker
which edge:
Text for Label2
Non-visible components
brickMoverTimer AccelerometerSensor1


Display Invisible Components in Viewer
5:09 PM
Car Game v. 1.0
Non-visible components
backgroundMusicPlayer obstacleMoverTimer carMoverTimer VibrationDetector
PowerupsAppearTimer ShieldTimer CrashSound ShieldSound LongMoo CrossingCowTimer
LifePowerSound GameTimer


Display Invisible Components in Viewer
5:09 PM
Screen1
Score---------------- Score Misses------------ Misses
Game Over!!!!
Non-visible components
Timer Player1 AccelerometerSensor1 grenadeSound EndGame

Lessons Learned.

As anticipated, the experience of teaching first-hand turned out to be an effective strategy for accumulating points of weakness (areas of improvement) for AppInventor on the Game Development front. Learning occurred primarily in one of two ways:

- A student asked a question along the lines of "How do you do *xyz* in AppInventor?" and the author was struck by the realization that AppInventor currently does not support operation *xyz*. If operation or entity *xyz* turned out to be a useful one for game development (i.e. if the need for it came up several times amongst the students during the course), the author noted it down on his list of "AppInventor Game Development limitations".
- If the author experienced frustration or difficulty in managing the class or in communicating a particular topic, he noted it down as an area that might possibly be improved by enhancements in his curriculum or teaching style.

The first bullet-point corresponds to AppInventor limitations and the second one corresponds to Pedagogical Challenges. Both topics are expounded below:

AppInventor Limitations:

1. **Event handling** for clicks (on buttons) and touches (on canvases) is imprecise since there is no separate event for touch-down and touch-released or button-pressed and button-released. There is only a single event that fires when the release happens. The result of this is that a developer cannot implement logic such as "move the car left while button is pressed, and stop the car once button is released", which is usually how such games work. It is unreasonable to expect the user to press the button repeatedly in order to keep the car object moving.

2. **Animation of sprites** is not supported, and by default, animated GIFs are not supported on Android. So, if a developer has three different frames/images for a Mario character that need to be displayed in succession to give the impression that Mario is walking, he/she would

need to flip through them manually. Automating this process would be useful.

3. **No support** for multi-touch input or **gesture detection**.

4. No support for having **sprites that are simple shapes** such as rectangles, ellipses, polygons, etc. If a developer wants to have a rectangular sprite, he/she would need to upload an image of a rectangle, which should not be necessary since a Rectangle is a simple and commonly used shape.

5. AppInventor **only supports collision detection with rectangular-bounds**, as opposed to pixel-by-pixel detection or polygon-based detection. The result is that for games where collision is being detected between two non-rectangular sprite objects, the results could be inaccurate on the corners.

6. Vertically / Horizontally **scrolling backgrounds** are not supported.

The author has addressed the first three of these limitations by implementing new AppInventor components or augmenting existing ones, and these implementations will be discussed in Chapter 4. The remaining three will be discussed in Chapter 6.

Pedagogical Challenges:

1. Students were spending too much time looking for images and sounds on the web, and in case they didn't find what they were looking for, they would spend hours editing images using Microsoft Paint or Adobe Photoshop. While this process of image-editing and resource-searching may play a role in developing the student's design sense and give the student a sense of ownership and creative license, it takes away from instruction time.

   - Solution: The author believes that the solution may be a hybrid approach that encourages students to search for resources on the web for *some* of the projects, but to pick from an existing Media Library for other projects.

2. Instead of requiring each student to build the same game for his/her Final Project in the course, the author allowed each student to build a game of

his/her choice from scratch. A side effect of this policy was that the topics that each student needed to learn in order to complete his/her game were different from those that others needed to learn. Furthermore, some students are faster at picking up new topics than others. As a result, the author often found himself in the difficult position of having to teach different topics to different groups at different paces, all in a limited amount of time. How does one instructor succeed in teaching all students in the course exactly what they wish to learn when the things they want to learn are all *different*?

- Solution: The author decided to make a set of 20 FlashCards containing mini-tutorials on various AppInventor Game Development concepts for his students. Now, when a student wants to learn a particular Game Development concept, he/she can check the flash-cards and attempt to teach himself / herself before asking the instructor to teach the topic. Therefore, even if five different students wanted to learn five different topics all at the same time, they would be able to do so by relying on five of the flash cards.

The "Media Library" and "Flashcards" discussed here are provided in Chapter 5, as a part of the Modified Curriculum.

Chapter 4

Modifications to AppInventor

Four modifications.

Chapter 3 identified six limitations of AppInventor that were discovered through the author's first-hand experiences teaching Game Development in India and the US. Three of those limitations (lack of touch-down and touch-up in event handling, lack of basic gesture recognition, and lack of easy-to-make animations) have been addressed in this thesis. In summation, the following four modifications have been made to AppInventor:

1. Canvas objects and sprite objects can now handle a "TouchDown" event and "TouchUp" event in addition to the "Touched" event.

2. AppInventor developers can now detect "fling" gestures on the Canvas object or sprite objects simply by handling the "Flung" event.

3. AppInventor developers can now easily animate images (e.g. make a walking man, or a jumping character) by uploading multiple frames of an animation via an AnimatedSprite component.

4. A new sensor, known as ProximitySensor, has been added. This sensor can detect when a human's hand or other object passes over the phone.

Details of these modifications follow.

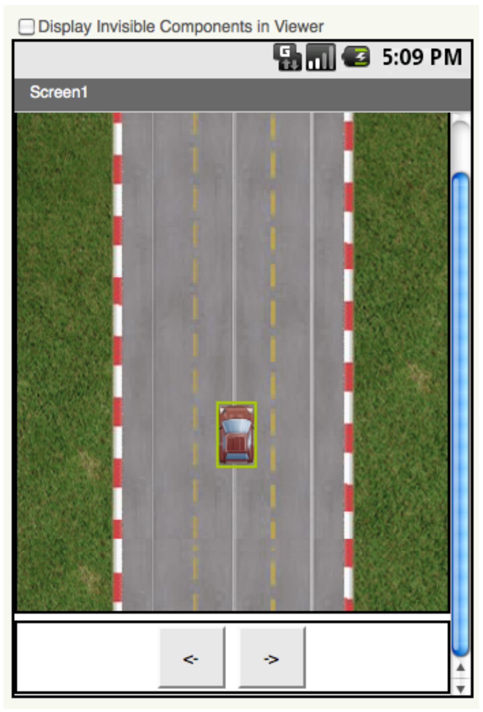
Precise handling of touch events.

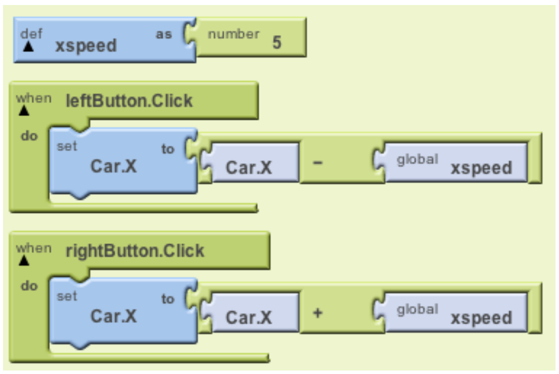
Imagine that you wish to make a simple car-game where the objective is to move a car left and right, avoiding obstacles coming on the road in the opposite direction.

You would like to be able to control the car using a left and right button; the car should move left while the left button is pressed, and right while the right button is pressed. If no buttons are pressed, the car should not move.

We can capture left/right button input from the user in two ways in AppInventor: either using Button components or using ImageSprite components.

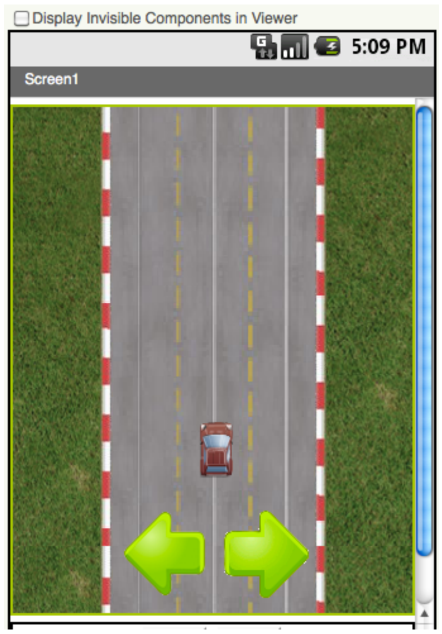
Using Button components, the project setup and Blocks Code would look like this:

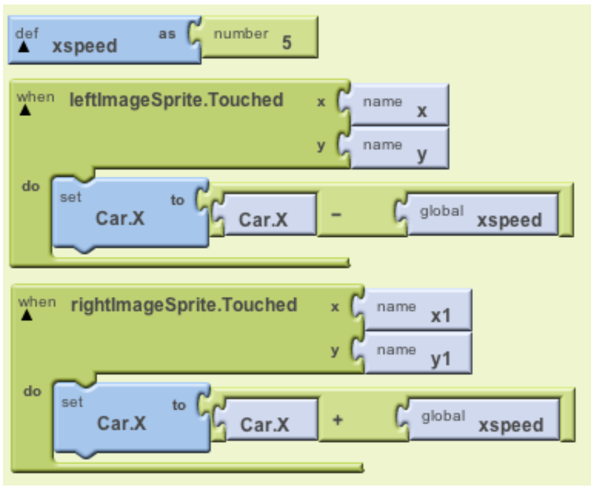




```
def xspeed as number 5

when leftButton.Click
do  set Car.X to  Car.X - global xspeed

when rightButton.Click
do  set Car.X to  Car.X + global xspeed
```

And using ImageSprite objects as buttons, the project setup and Blocks Code would look like this:



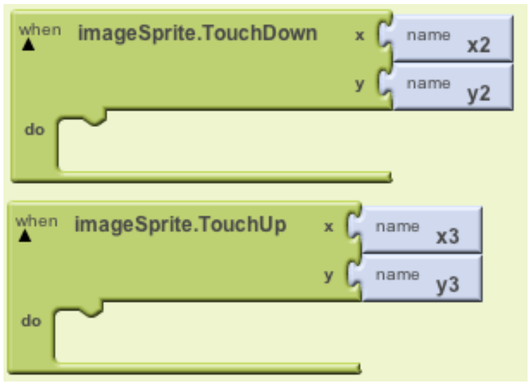


```
def xspeed as number 5

when leftImageSprite.Touched  x name x
                              y name y
do  set Car.X to  Car.X - global xspeed

when rightImageSprite.Touched  x name x1
                               y name y1
do  set Car.X to  Car.X + global xspeed
```

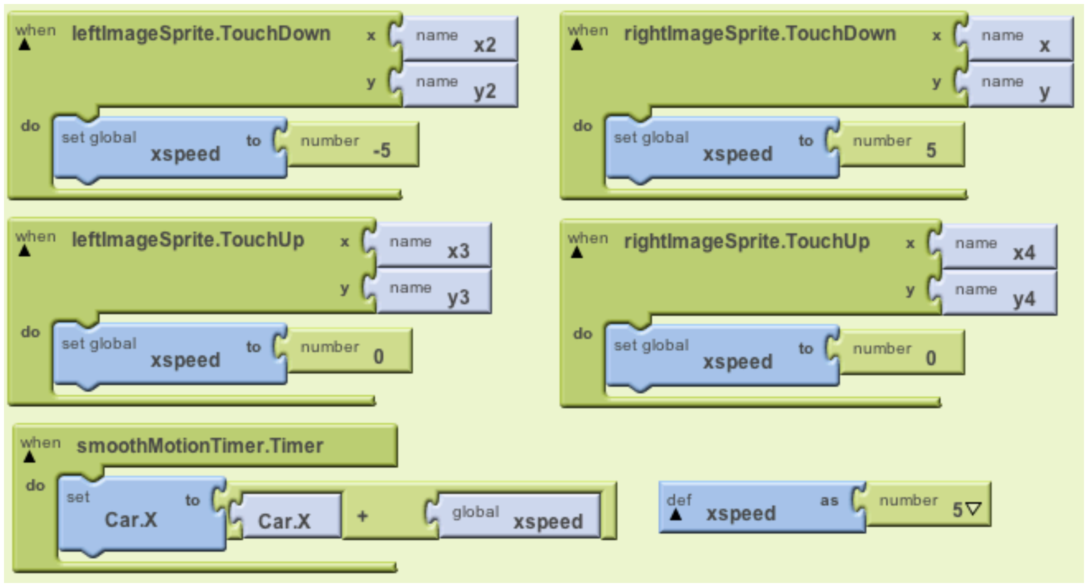
However, in both of these approaches, the game player would have to keep tapping the phone to keep the car moving, a task that can quickly become irritating and tiresome.

To cause the car to move smoothly *while* a button is pressed requires knowledge of when touch-down and touch-up occurred on the button.

The modified version of AppInventor provides these two event-handlers, fortunately, both for the Canvas component as well as for all types of sprite components (e.g. ImageSprite). The interface for these two event handlers looks identical to that of Touched, and therefore should be comprehensible to developers:


when imageSprite.TouchDown x name x2 y name y2
do
when imageSprite.TouchUp x name x3 y name y3
do

The blocks-code implementation of the car game using TouchUp and TouchDown would look as follows:


when leftImageSprite.TouchDown x name x2 y name y2
do set global xspeed to number -5
when rightImageSprite.TouchDown x name x y name y
do set global xspeed to number 5
when leftImageSprite.TouchUp x name x3 y name y3
do set global xspeed to number 0
when rightImageSprite.TouchUp x name x4 y name y4
do set global xspeed to number 0
when smoothMotionTimer.Timer
do set Car.X to Car.X + global xspeed
def xspeed as number 5

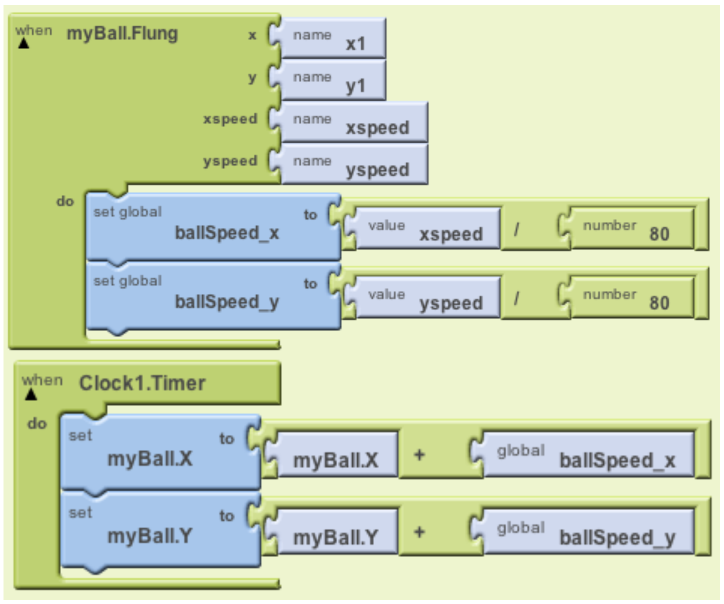
]

Fling gesture detection.

AppInventor now has the ability to detect fling gestures, which are quick finger-motions that resemble swipes.

Just as Canvas components and sprite components can both detect Touched, TouchUp, and TouchDown events, these two components can also detect a Flung event. It is up to the developer to decide which component should handle the Fling.

If a developer wishes to detect *any* fling gesture that takes place anywhere on the canvas, then he/she should handle the "Canvas.Flung" event. On the other hand, if he/she only cares about situations when a particular sprite is flung, then he/she should handle the Flung event of on that sprite. Below is an example of how to detect flings on a "Ball" component:



```
when myBall.Flung
     x       name x1
     y       name y1
     xspeed  name xspeed
     yspeed  name yspeed
do   set global ballSpeed_x to  value xspeed / number 80
     set global ballSpeed_y to  value yspeed / number 80

when Clock1.Timer
do   set myBall.X to  myBall.X + global ballSpeed_x
     set myBall.Y to  myBall.Y + global ballSpeed_y
```

Note that the Flung event handler block provides xspeed and yspeed as arguments. These values represent how fast the finger was swiped on the screen in the x and y directions, respectively. In the above example, the division by 80 (an arbitrary number) is performed in order to scale down the fling x and y speeds so that they are on the order of pixels per second and are reasonable, given the screen dimensions of most Android phones on the market.

The AnimatedSprite component.

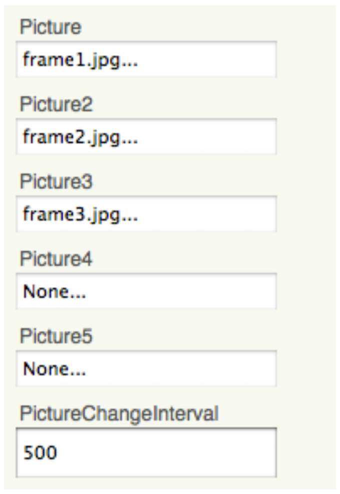
The AnimatedSprite component is essentially an ImageSprite that stores multiple images and automatically flips through them every few milliseconds (determined by the "PictureChangeInterval" property of the component).

AnimatedSprite components fall within the same Component cateogry as ImageSprite and Ball components, and can be dragged onto the Canvas in exactly the same way. The difference arises in the picture-uploading process: while for ImageSprite components the developer can only upload one picture, the AnimatedSprite allows you to upload up to 5 pictures and it automatically cycles through them.

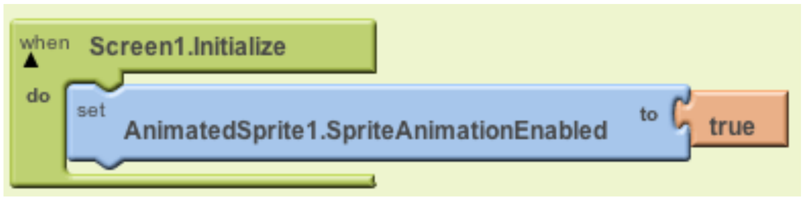
To change how many milliseconds the component should wait before changing the picture, the developer should change the PictureChangeInterval property, either from the Properties panel in the designer view, or from the blocks code.

To turn "on" the animation of the sprite, the developer should set the "SpriteAnimationEnabled" property to true, either from designer view or blocks code.

Here's an example of how an AnimatedSprite component can be configured and used:



Picture
frame1.jpg...
Picture2
frame2.jpg...
Picture3
frame3.jpg...
Picture4
None...
Picture5
None...
PictureChangeInterval
500



when Screen1.Initialize
do set AnimatedSprite1.SpriteAnimationEnabled to true

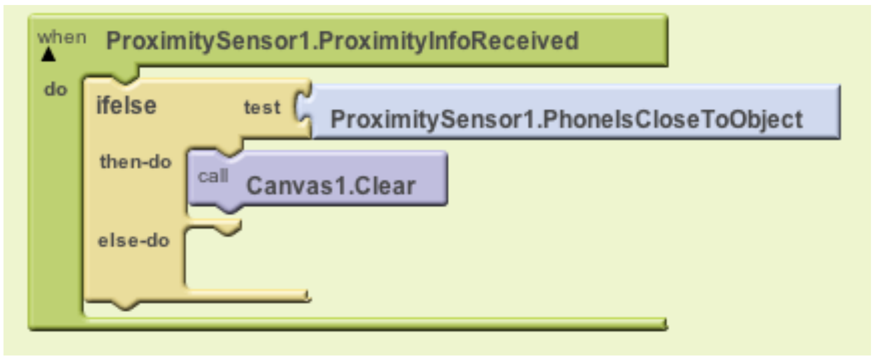
The ProximitySensor component.

The proximity sensor is a piece of hardware that is built into most Android phones and has the ability to detect (by sensing ambient light) whether or not the phone is close to another object (e.g. if the phone is next to somebody's ear). However, until now, this sensor did not exist within AppInventor.

Generally, a ProximitySensor can prove useful if a developer wishes to implement some logic that reacts to the phone being close to an object (as an example, imagine a humorous app that yells "boo!" as soon as a user puts the phone next to his/her ear). In the world of Game Development and Visual programming however, the proximity sensor can be handy since it provides yet another sort of 'gesture' recognition -- specifically, for the hand-swipes-over-the-phone gesture.

If a developer is building the PaintPot application from the AppInventor online tutorials, then he/she can get rid of the "Clear" button and instead clear the canvas when the user moves his/her hand across the top of the phone (without touching the phone) in a swiping motion. Likewise, within the context of a game, this above-the-phone swiping motion could be used as a cue to restart the game, or to pause/resume the game.

Using the sensor is simple. The developer simply handles the "ProximityInfoReceived" event, and then within that event handler, checks whether the value of the "PhoneIsCloseToObject" property is true or false.



```
when ProximitySensor1.ProximityInfoReceived
do   ifelse   test ProximitySensor1.PhoneIsCloseToObject
     then-do  call Canvas1.Clear
     else-do
```

Chapter 5

Modified Curriculum

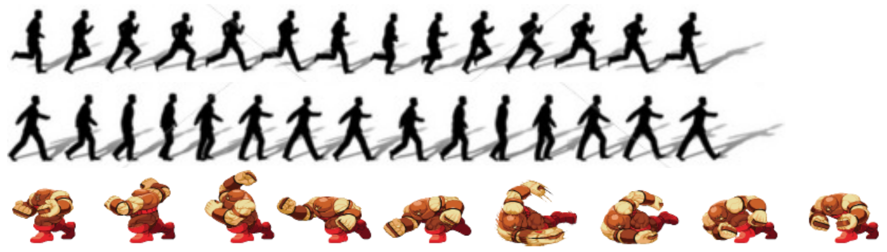
Curriculum Overview

	Planned Activities	Concepts Learned
Day 1 (8 hrs)	Ice Breakers Playing with Android Phones AppInventor Setup Overview of AppInventor's interface Project 1: HelloPurr and Media Player Project 2: Calculator	Sound, Button, Screen, event handling, calling procedures, connecting to device, Player, screen arrangements, Label, TextBox, getting and setting property values, if-else statements, packaging apps to phone
Day 2 (8 hrs)	Project 3: Drawing Program Project 4: Balls, Sprites, and Motion of Objects 5 ways to make things move (automatically; using Clock, using Buttons, using Accelerometer, using Fling gestures)	Canvas, 'dragged' and 'touched' events, drawing lines and circles, colors, ProximitySensor, Ball, ImageSprite, edge-detection, bouncing ball, custom variables, custom procedures, Clock, Accelerometer, Fling gesture, Using ImageSprites as Buttons
Day 3 (8 hrs)	Project 5: Car on a Road Project 6: Paper Prototyping Begin final project (game of student's choice)	collision detection, random-number generation, making multiple objects move simultaneously using Clock,
Day 4 (8 hrs)	Project 7: Fun with Animations Continue final project (game of student's choice)	Students will learn about AnimatedSprite objects. The remainder of the instruction on Day 4 will be one-on-one and highly customized to each student's or group's needs.
Day 5 (8 hrs)	Publishing AppInventor apps to Google Play Continue final project (game of student's choice) Presentations of final projects	Google Play developer registration process, downloading AppInventor apps as .apk files, uploading .apk files to Google Play. The remainder of the learning on Day 5 will be customized to each student's or group's needs.

Media Library

When asked to find images or audio "on the web" for their games, students often get distracted or spend too long searching for the *perfect* image, or in editing images on Photoshop. So, we provide this library of good-looking and useful graphics and audio for the students to use in their games, to make sure they stay focused on the real purpose of this goal -- to learn programming.

Images for Animation

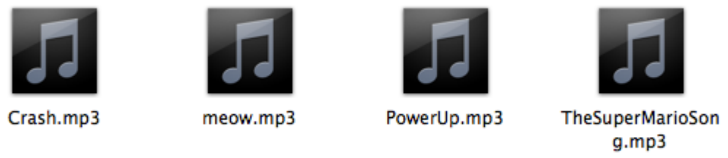


Images for Game Objects

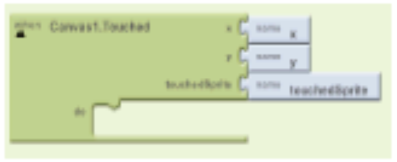
Images for Buttons

Background Images

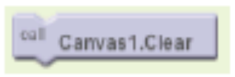
Audio Files



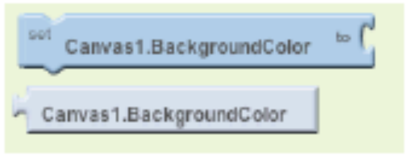
AppInventor Flash Cards



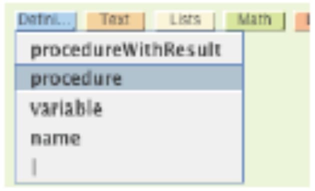
Events are actions that can be detected and responded to.



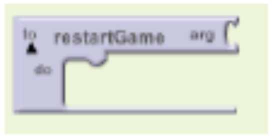
Procedures are operations that can be performed by components, e.g. "Clear" is an operation on Canvas objects.



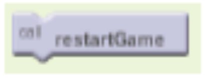
Properties are qualities of an object that can be read and modified.



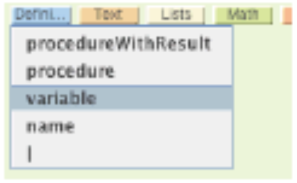
Click anywhere on the screen, and a colorful menu should pop up. On the "Define..." tab, click "procedure" or "procedureWithResult".



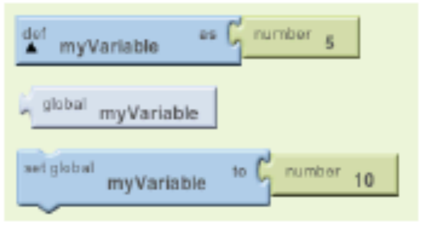
You may give your procedure a name, e.g. "restartGame", and then place the code that should be executed when the procedure is run inside the block.



You may call the procedure you have defined from within your blocks code.

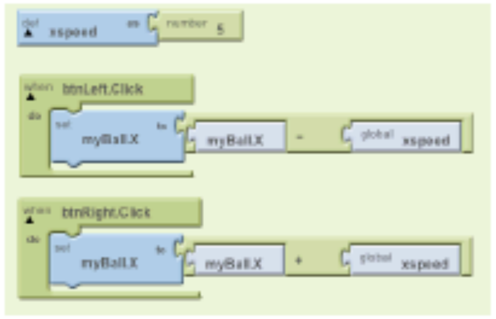


Click anywhere on the screen, and a colorful menu should pop up. On the "Define..." tab, click "variable".



You must define your variable using a "def" block. After that, the value of the variable can be read or modified as many times as is necessary.

Declare a variable that represents the 'speed' of the sprite, i.e. how many pixels to move each time the button is pressed. Then use this variable to update the position of the sprite in the buttons' click handlers.



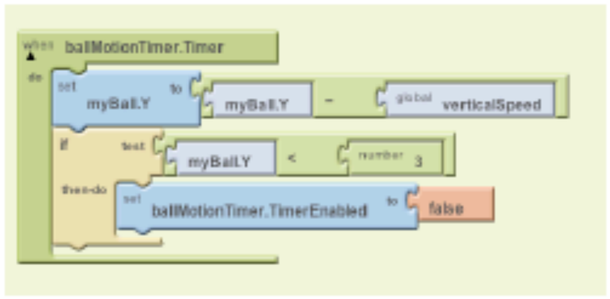
Implementing Jumping and Gravity

Moving Sprites with Clocks

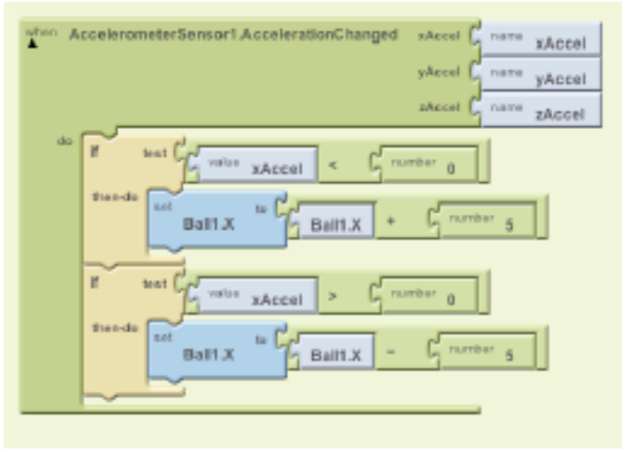
Using AnimatedSprite Component

Moving Sprites with Accelerometer

Drag a Clock component onto the project. Add a handler for the "Timer" event of the Clock, and inside that handler, include code that you would want to execute every time the timer fires.

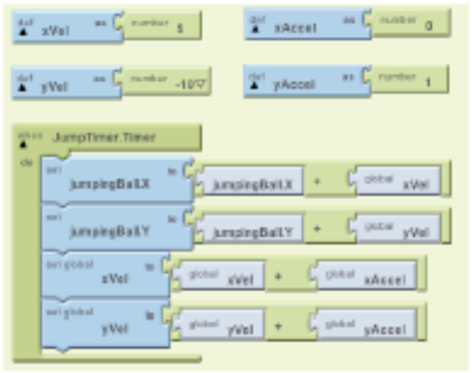


```
when ballMotionTimer.Timer
do  set myBall.Y to  myBall.Y - global verticalSpeed
    if test  myBall.Y < number 3
    then-do  set ballMotionTimer.TimerEnabled to false
```



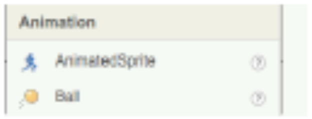
```
when AccelerometerSensor1.AccelerationChanged  xAccel name xAccel
                                               yAccel name yAccel
                                               zAccel name zAccel
do  if test  value xAccel < number 0
    then-do  set Ball1.X to  Ball1.X + number 5
    if test  value xAccel > number 0
    then-do  set Ball1.X to  Ball1.X - number 5
```

Gravity is represented as a positive y-acceleration value and the intial y-velocity is a negative number. This is because downwards is positive and upwards is negative in the coordinate system of AppInventor.

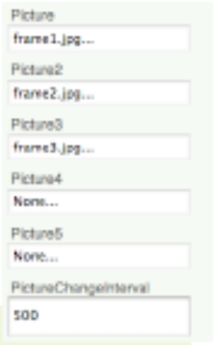


```
def xVel as number 5          def xAccel as number 0
def yVel as number -10        def yAccel as number 1

when JumpTimer.Timer
do  set jumpingBall.X to  jumpingBall.X + global xVel
    set jumpingBall.Y to  jumpingBall.Y + global yVel
    set global xVel to  global xVel + global xAccel
    set global yVel to  global yVel + global yAccel
```

step 1: add an AnimatedSprite component to the project.

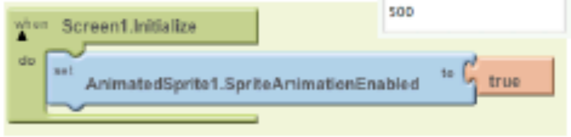


step 2: upload up to 5 images for the component through the Properties panel in the designer.



Picture	frame1.jpg...
Picture2	frame2.jpg...
Picture3	frame3.jpg...
Picture4	None...
Picture5	None...
PictureChangeInterval	500

step 3: Set SpriteAnimationEnabled to true when app starts.



```
when Screen1.Initialize
do  set AnimatedSprite1.SpriteAnimationEnabled to true
```

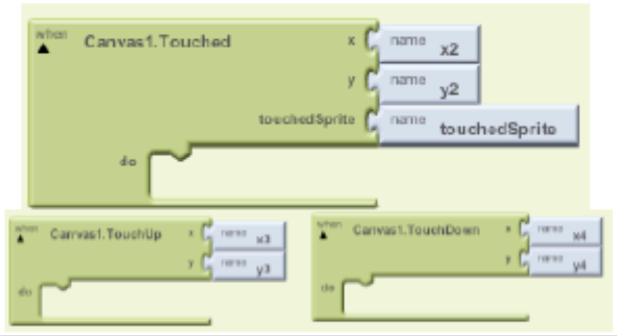
Detecting if Phone is Close to Objects

Touch Events on Canvas and Sprites

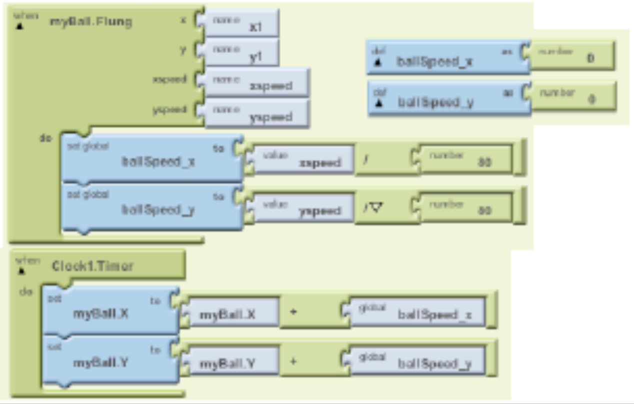
Playing Sounds in your App

Detecting Fling Gestures

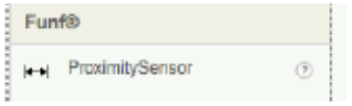
There are three types of touch events that can be handled for Canvas and Sprite objects. The "TouchDown" event is triggered as soon as a user's finger touches down on the object, "TouchUp" is triggered when the finger lifts off the object, and "Touched" is triggered when the object is tapped (when TouchDown and TouchUp happen quickly in succession).



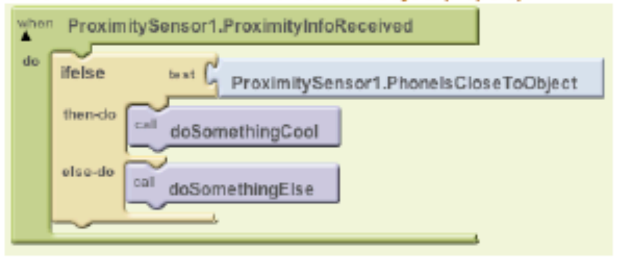
A fling is a quick swipe gesture on the screen. The xspeed and yspeed arguments indicate how fast the flung object should move in the x and y directions.


when myBall.Flung
x name x1
y name y1
xspeed name xspeed
yspeed name yspeed
do
set global ballSpeed_x to value xspeed / number 80
set global ballSpeed_y to value yspeed / number 80
def ballSpeed_x as number 0
def ballSpeed_y as number 0
when Clock1.Timer
do
set myBall.X to myBall.X + global ballSpeed_x
set myBall.Y to myBall.Y + global ballSpeed_y

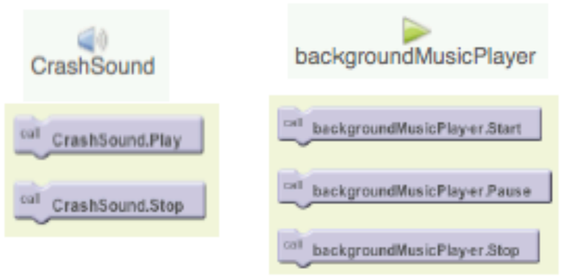
step 1: Drag a ProximitySensor object onto the project.



step 2: In the handler for ProximityInfoReceived event, check the value of the PhoneIsCloseToObject property.


when ProximitySensor1.ProximityInfoReceived
do
ifelse
test ProximitySensor1.PhoneIsCloseToObject
then-do call doSomethingCool
else-do call doSomethingElse

There are two components in AppInventor that can play audio. **Sound** components are for playing short audio files, whereas the **Player** component is for playing longer tracks, e.g. background music.


CrashSound
call CrashSound.Play
call CrashSound.Stop
backgroundMusicPlayer
call backgroundMusicPlayer.Start
call backgroundMusicPlayer.Pause
call backgroundMusicPlayer.Stop

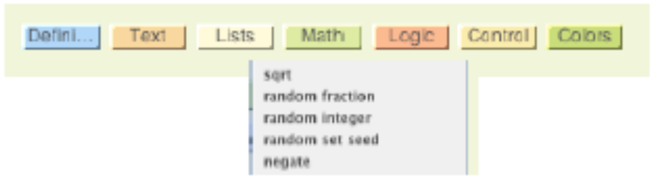
Layouts and Centering Components

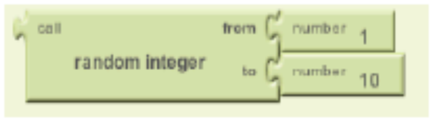
Generating Random Numbers

Collision Detection

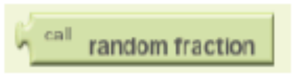
Detecting if Phone is Shaking

Click anywhere on the screen, and a colorful menu should pop up. On the "Math..." tab, click "random fraction" or "random integer".



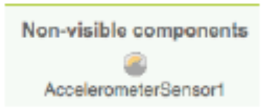


generates a random number between the two specified numbers (inclusive)

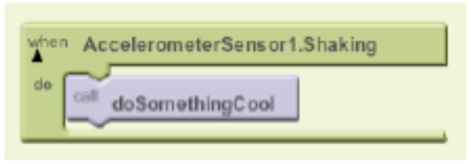


generates a random decimal value between 0 and 1.

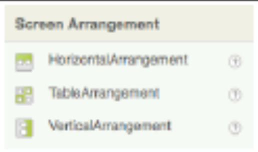
Step 1: drag an Accelerometer onto the project.



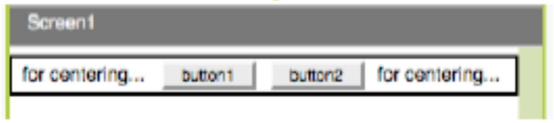
Step 2: in the blocks code, handle the "Shaking" event of the Accelerometer component.



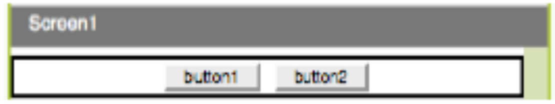
Step 1: Choose from 3 available screen arrangements.



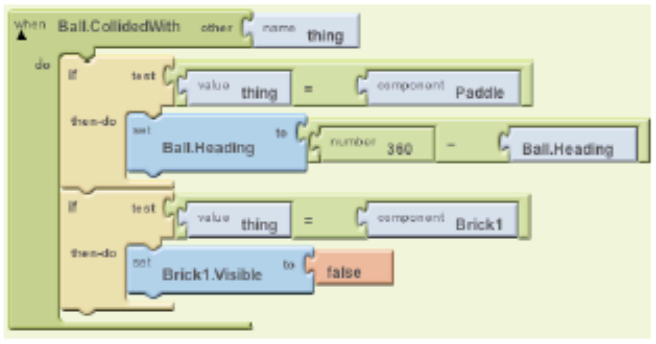
Step 2: Drag multiple components into the arrangement. In order to center them, drag a Label component onto both ends of the arrangement.



Step 3: set the Width of each Label to "Fill parent" and the TextColor to "None". Set the Width of the entire arrangement to "FillParent" as well.

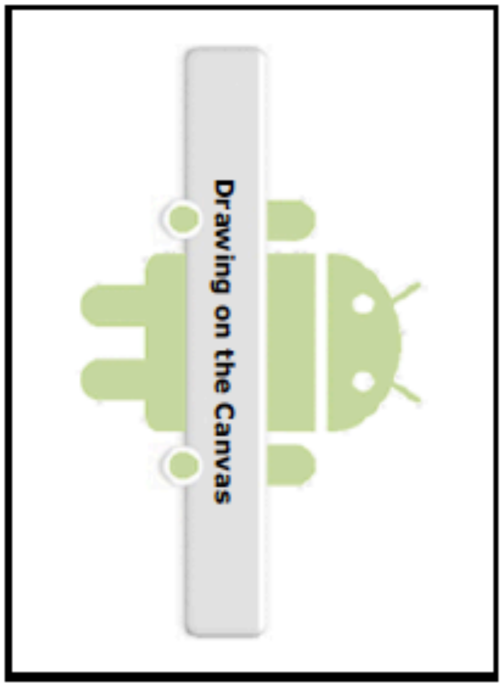


For a game such as brick breaker (where you direct a ball using a paddle, trying to hit bricks), you can check collision by handling the "CollidedWith" event on the Ball component:


when Ball.CollidedWith other name thing
do
if test value thing = component Paddle
then-do set Ball.Heading to number 360 - Ball.Heading
if test value thing = component Brick1
then-do set Brick1.Visible to false

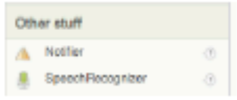
Creating your own Colors

Speech Recognition

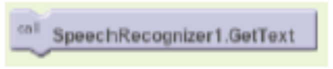

Drawing on the Canvas

Working with Lists of Components

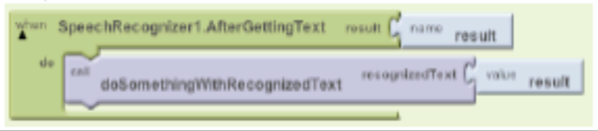
Step 1: Drag a "SpeechRecognizer" component from the "Other stuff" category onto your project.



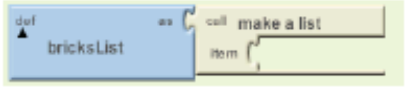
Step 2: In the blocks code, call "GetText" on the component whenever you'd like to read speech input from the phone.



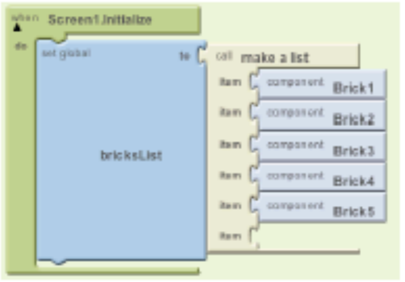
Step 3: Handle the "AfterGettingText" event, which will be triggered once speech as been received and recognized by the component.


when SpeechRecognizer1.AfterGettingText result
name result
do
call doSomethingWithRecognizedText recognizedText
value result

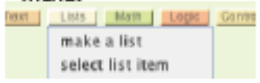
Step 1: Define a new variable and set its value to an empty list, which you can create by calling the "make a list" procedure.



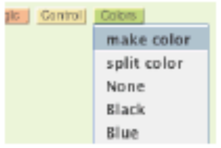
Step 2: In the screen's Initialize event handler, set the value of the variable to a new list that contains the components of your choice.



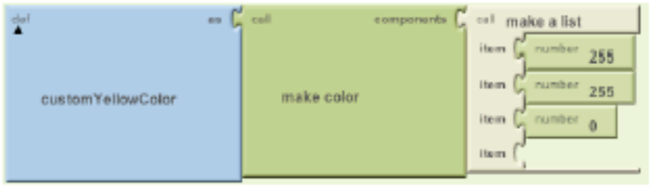
Note: You can access the "make a list" procedure by clicking anywhere on your blocks editor, and choosing the "Lists" tab from the colorful menu.

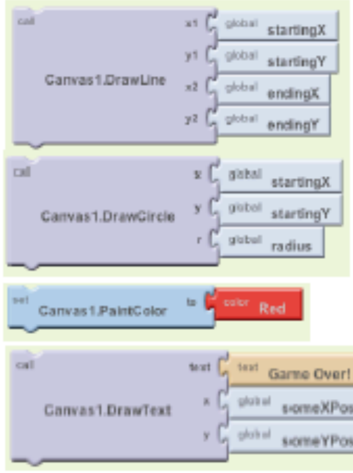


Colors in AppInventor are represented by 3 numerical values known as RGB (red, green, blue). Each value must be between 0 and 255.



New colors are constructed by mixing different amounts of Red, Green, and Blue. Yellow, for example, is Red + Green, so its RGB values will be R = 255, G = 255, B = 0. In AppInventor, use the "make color" procedure and provide RGB values as elements of a list.





You can draw circles, lines, text, and points onto the canvas using the built-in draw procedures of Canvas components. These procedures take x and y values as arguments. Keep in mind that (0,0) is at the top-left of the screen. You can change the drawing color at any time by setting the value of the PaintColor property.

Day 1

Begin with icebreakers (nicknames, fun facts, what you want to learn, why you enrolled in course, etc.)

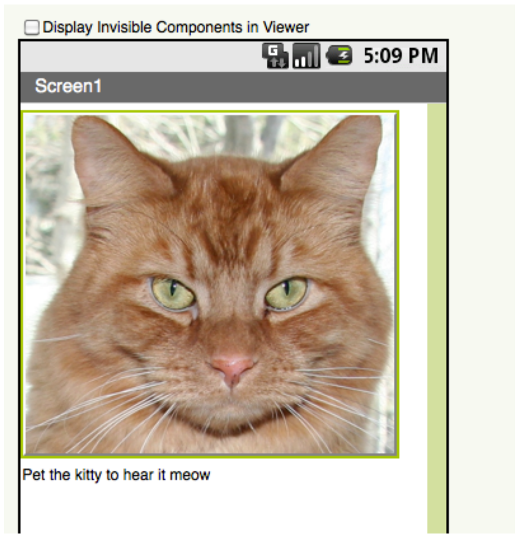
Hand out phones and connector cables, allow students to play around with and get a feel for the devices. Attract their attention to the Google Play application (formerly known as the Market) on the Android phones and excite them about the prospect of publishing their own application to Google Play (to be discussed on Day 5).

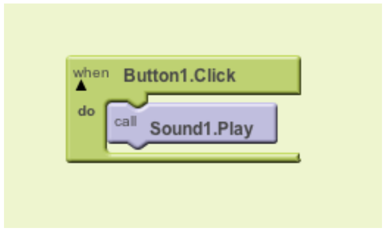
Ask students to perform AppInventor setup, following the instructions at http://beta.appinventor.mit.edu/learn/setup/.

Walk them through the basics of AppInventor's interface (designer view, palette, properties panel, visible vs. non-visible components, and blocks editor).

Project 1: HelloPurr and MediaPlayer

Start with simple "HelloPurr" app to introduce **Sound**, **Button**, **Screen** components and **event handling**.

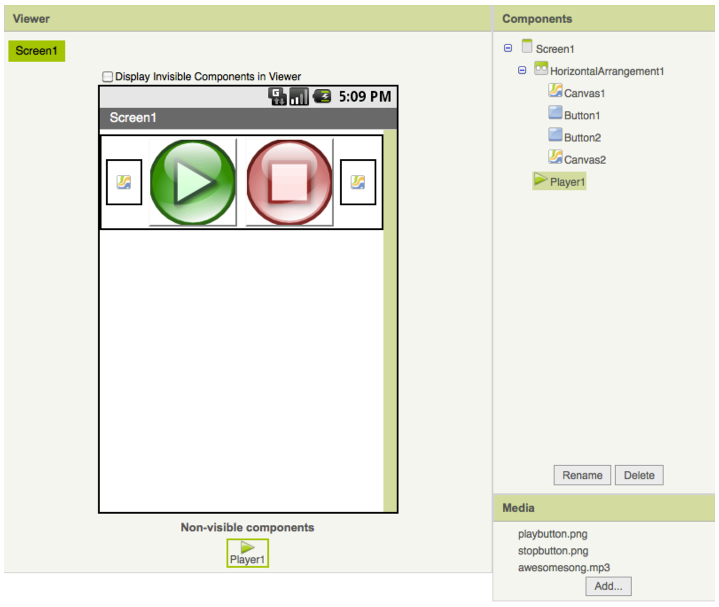

Display Invisible Components in Viewer
5:09 PM
Screen1
Pet the kitty to hear it meow

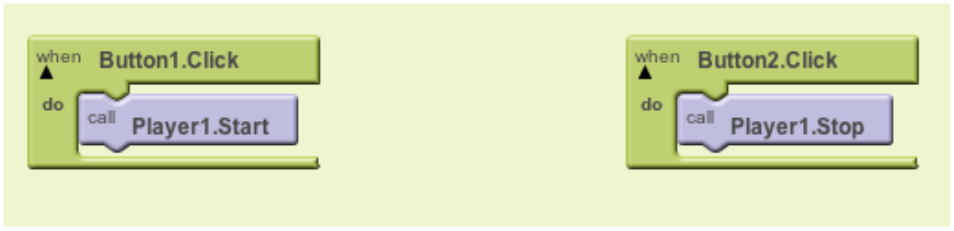


Describe the difference between **events** and **procedures** (also known as functions) in the context of HelloPurr. Button1.**Click** is a handler for an **event**, whereas Sound1.**Play** is a procedure.

Explain how to **connect to device** from the BlocksEditor, so that students can test their HelloPurr apps on their devices.

Students should partner up and modify their HelloPurr app to turn it into a Media Player with play and stop buttons. Also, introduce the **Player** component here, which can play long songs, whereas the Sound component from the HelloPurr app could only play short sounds.
Introduce the concept of layouts + **screen arrangements**. Show the students how to put the Play button and the Stop button on the same line by inserting both of them into a **HorizontalArrangement** component. Show the students how to *center* the buttons by adding empty components (such as Canvas or Label) on the left and right side of the HorizontalArrangement so that they serve as spacers.

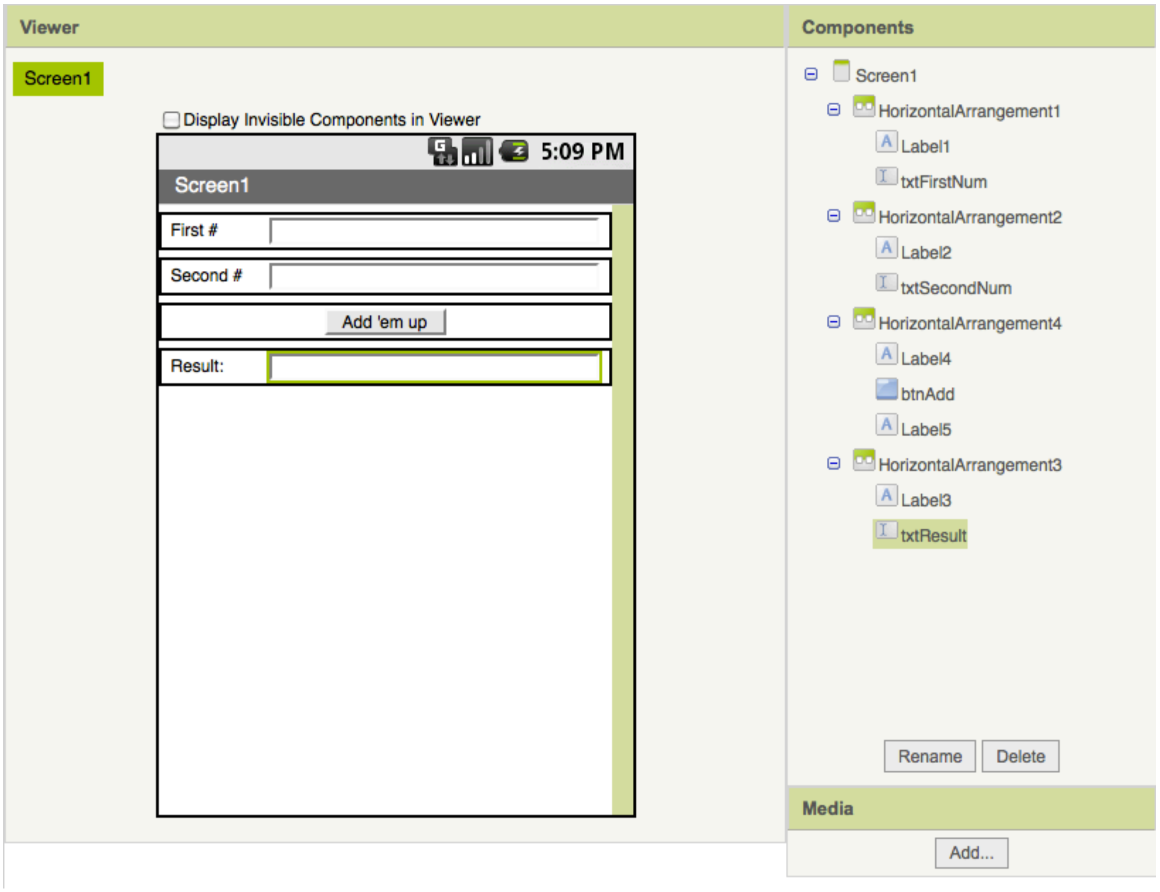




Extra Credit: Add a pause/resume button, in addition to the play and stop buttons. Student may choose to have the 'resume' state of the pause-resume button look different from the 'pause' state -- if this is the case, teach them how to change the button's background image when the button is pressed, by setting the "Picture" property to a string representing the filename of the new button image.

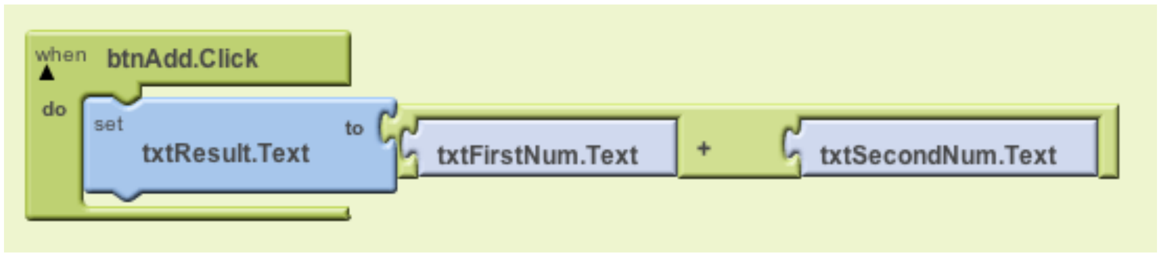
Project 2: Calculator

Introduce students to **Textbox** and **Label** components, and ask them to use these along with HorizontalArrangements and Buttons to produce the interface for a two-number adder that looks as follows.

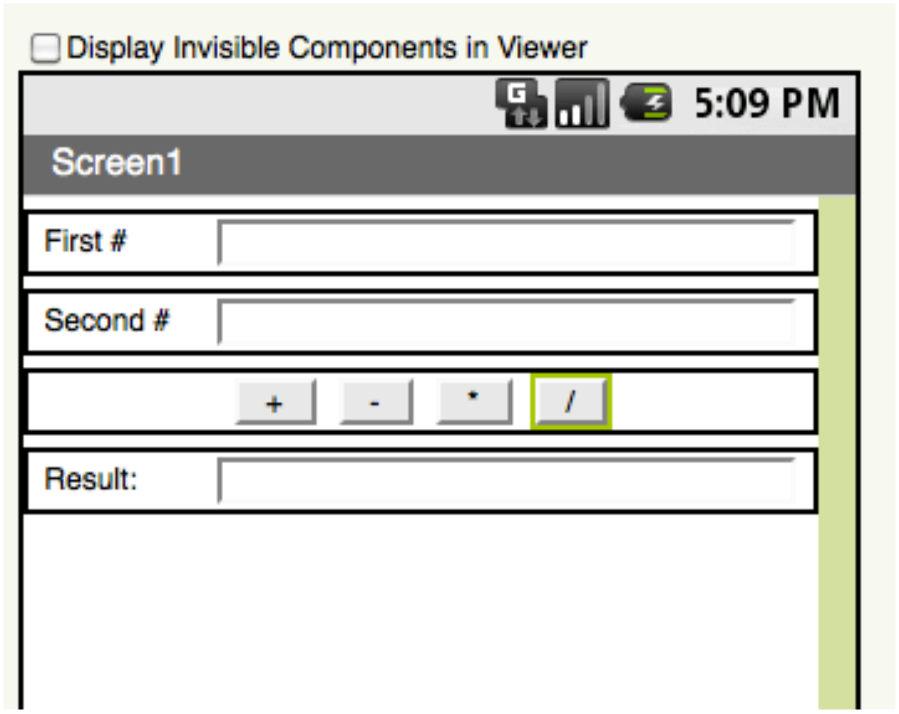


The goal of this adder is to add the two numbers typed in by the user and to indicate their sum in the result text box when the 'add' button is pressed.

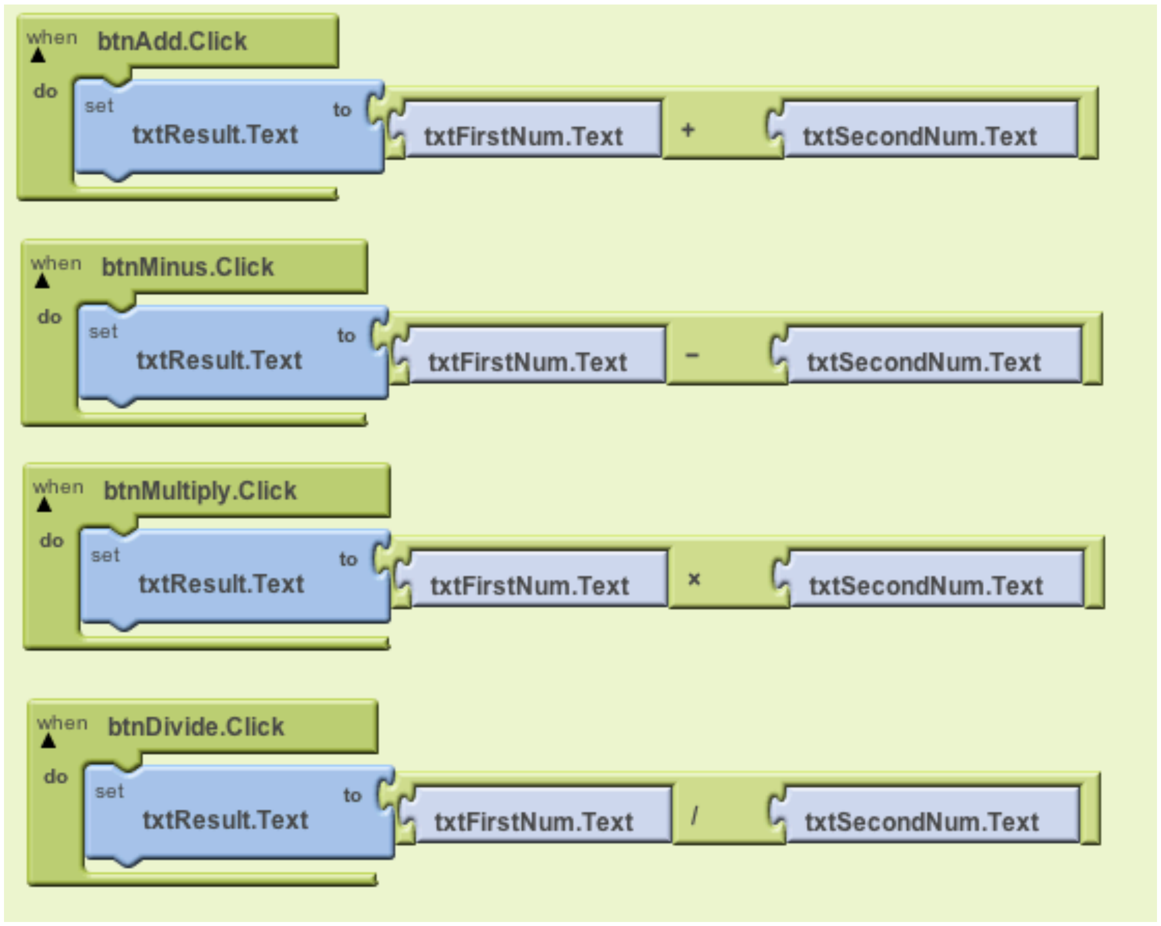
As the students implement this functionality in the Blocks Editor, they will learn about **properties** of components, and will learn to get and set their value.



Students should work individually to modify this adder and turn it into a calculator that can do all four operations on two numbers. The students should use screen arrangements to make this look nice, see below for a screenshot:

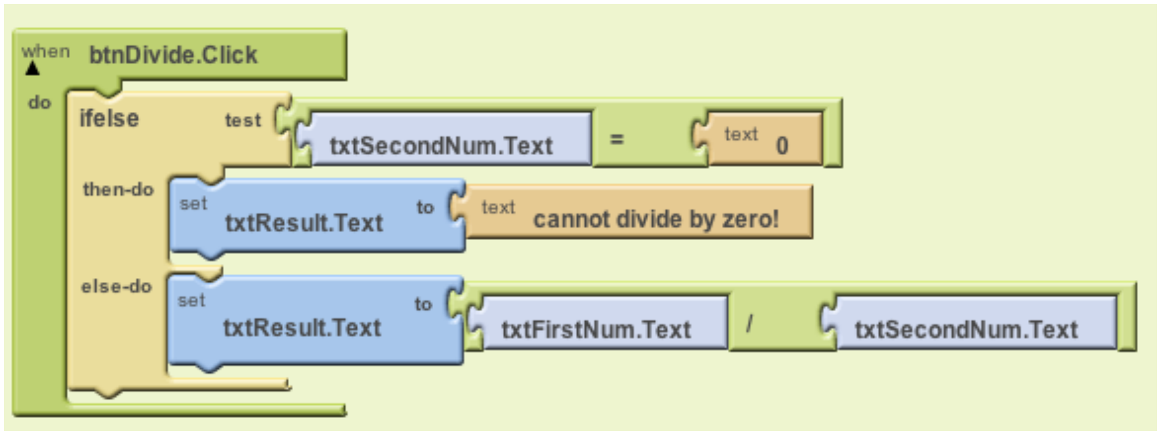


After the students have implemented the interface, ask them to follow the model of the adder they previously constructed to implement the subtraction, multiplication, and division operations.



```
when btnAdd.Click
do  set txtResult.Text to  txtFirstNum.Text + txtSecondNum.Text

when btnMinus.Click
do  set txtResult.Text to  txtFirstNum.Text - txtSecondNum.Text

when btnMultiply.Click
do  set txtResult.Text to  txtFirstNum.Text × txtSecondNum.Text

when btnDivide.Click
do  set txtResult.Text to  txtFirstNum.Text / txtSecondNum.Text
```

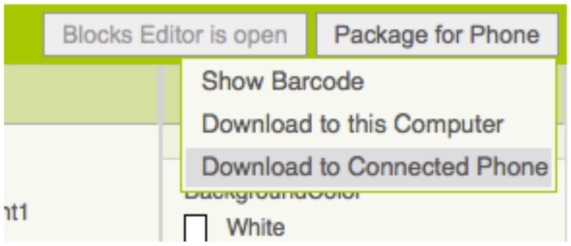
Next, teach students about basic control structures, specifically **if-else statements**. Ask students to use if statements to prevent division by zero. Here is one way of doing this:



```
when btnDivide.Click
do  ifelse  test  txtSecondNum.Text = text 0
            then-do  set txtResult.Text to  text cannot divide by zero!
            else-do  set txtResult.Text to  txtFirstNum.Text / txtSecondNum.Text
```

Extra Credit: Ask students that finish early to style their calculator by changing background color / images of the screen, buttons, textboxes, or other components.

Once all students are caught up, teach how to **package** appinventor apps to the phone so that the app can run on the phone without requiring the phone to remain connected to the computer. Ask students to package their calculators and their MediaPlayer apps and to download these to their Android devices.

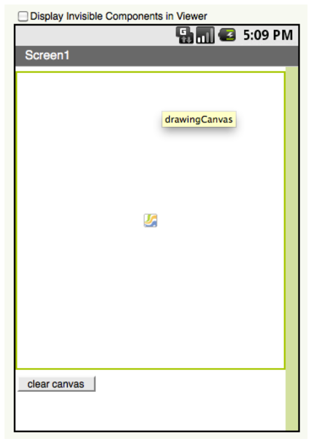


Day 2:

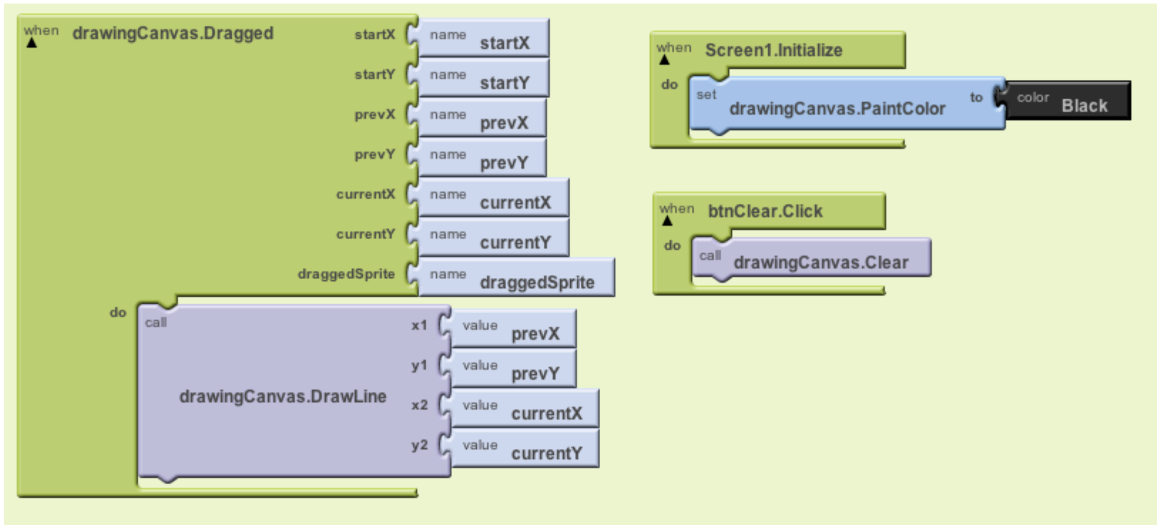
Project 3: Drawing Program

Introduce **Canvas** component as the fundamental container component for graphics (drawings, images, sprites, etc.).

Build a simple drawing application where user drags on the canvas to draw lines and can clear canvas by clicking the clear button.

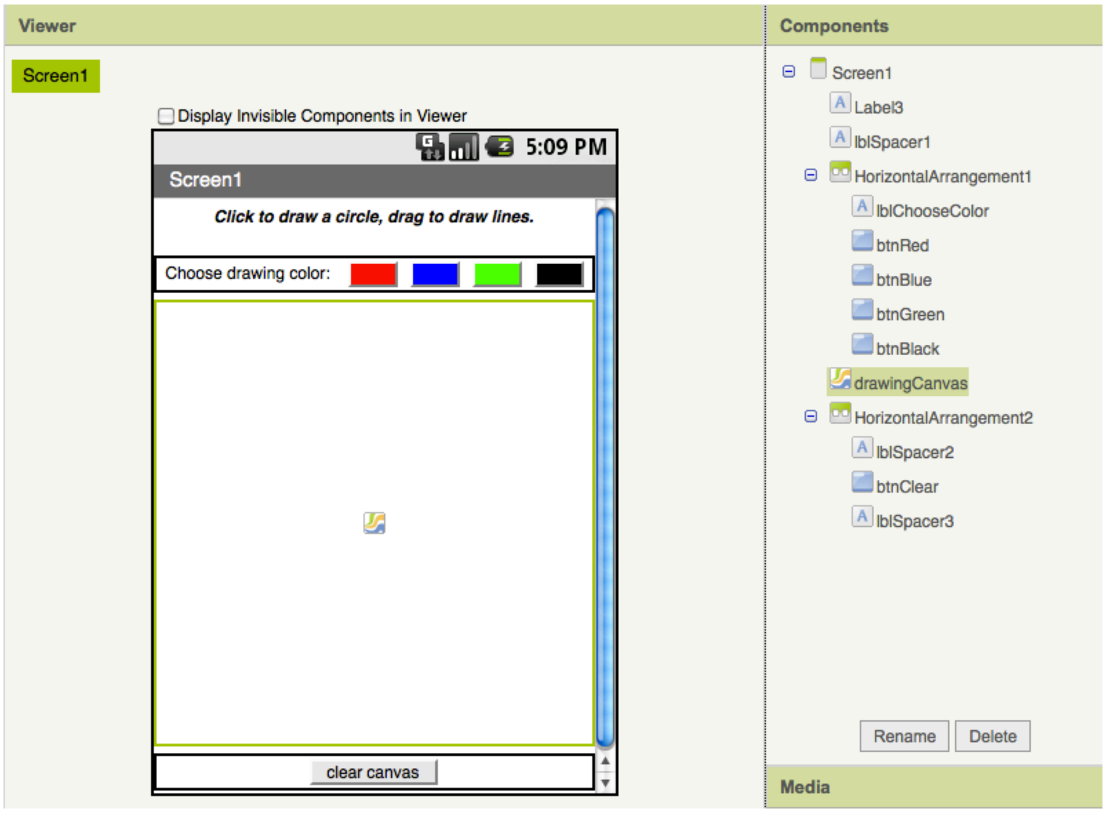


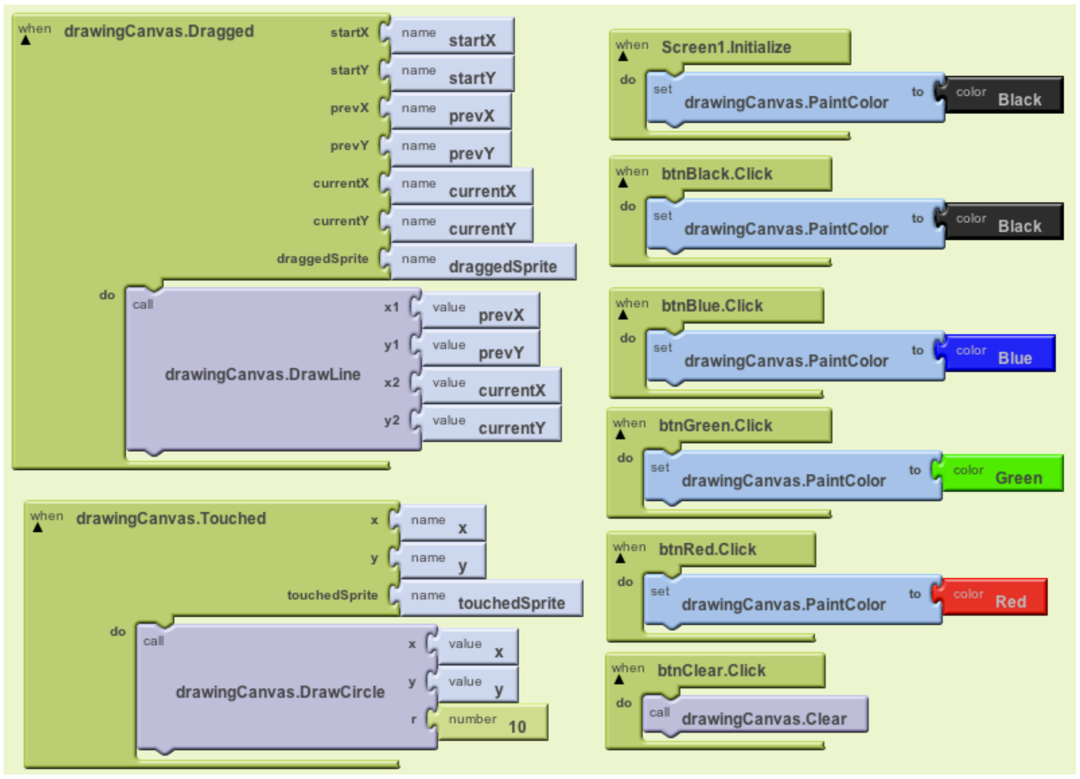
Show how to set the drawing **color** of canvas when screen (application) gets initialized, and how to handle **dragged** events.



```
when drawingCanvas.Dragged
    startX        name startX
    startY        name startY
    prevX         name prevX
    prevY         name prevY
    currentX      name currentX
    currentY      name currentY
    draggedSprite name draggedSprite
  do call drawingCanvas.DrawLine
        x1 value prevX
        y1 value prevY
        x2 value currentX
        y2 value currentY

when Screen1.Initialize
  do set drawingCanvas.PaintColor to color Black

when btnClear.Click
  do call drawingCanvas.Clear
```

Ask students to modify the drawing program to include buttons to set the drawing color of the application, and to draw circles in response to **touched** events.



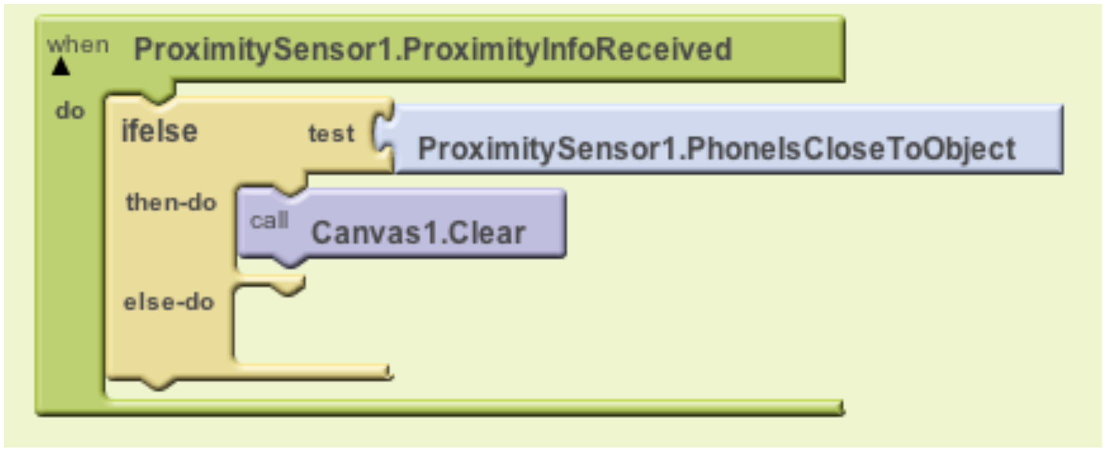


```
when drawingCanvas.Dragged
    startX name startX
    startY name startY
    prevX name prevX
    prevY name prevY
    currentX name currentX
    currentY name currentY
    draggedSprite name draggedSprite
do
    call drawingCanvas.DrawLine
        x1 value prevX
        y1 value prevY
        x2 value currentX
        y2 value currentY

when drawingCanvas.Touched
    x name x
    y name y
    touchedSprite name touchedSprite
do
    call drawingCanvas.DrawCircle
        x value x
        y value y
        r number 10

when Screen1.Initialize
do  set drawingCanvas.PaintColor to color Black

when btnBlack.Click
do  set drawingCanvas.PaintColor to color Black

when btnBlue.Click
do  set drawingCanvas.PaintColor to color Blue

when btnGreen.Click
do  set drawingCanvas.PaintColor to color Green

when btnRed.Click
do  set drawingCanvas.PaintColor to color Red

when btnClear.Click
do  call drawingCanvas.Clear
```

Once students finish the drawing program, teach them how to clear the canvas by sensing hand motions instead of relying on button-clicks.

Introduce **ProximitySensor**, and explain how we can detect when a user's hand is close to the proximity sensor on the phone and how we can use this to detect a general swiping motion of the hand above the phone.

The students should drag a ProximitySensor from the Funf component category onto their project, and should include the following in their Blocks code:



```
when ProximitySensor1.ProximityInfoReceived
do
    ifelse test ProximitySensor1.PhoneIsCloseToObject
    then-do
        call Canvas1.Clear
    else-do
```

Project 4: Balls, Sprites, and Motion of Objects

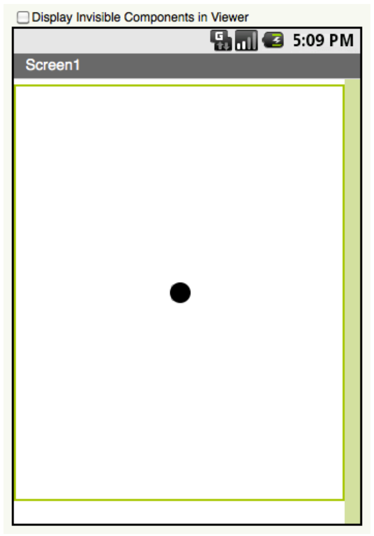
Introduce the **Ball** component as one of two Sprite components in AppInventor that have the ability to move around on a canvas. The second sprite component is **ImageSprite**, which is similar to a ball in that it can be moved around on a canvas, but it is visually represented by a custom Image that is uploaded by the AppInventor developer.

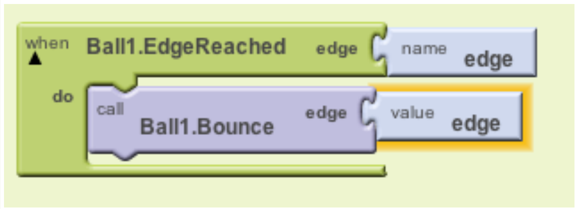
There are many ways to make Balls and ImageSprites move, but in this course we'll discuss the four of them (listed below) that are most applicable to game development:

1) Automatically (using built-in properties)
2) Using buttons
3) Using the AppInventor Clock component
4) Using Accelerometer
5) Using Fling gesture-detection

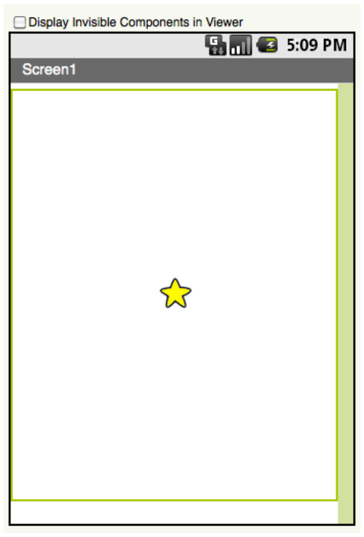
Motion Method 1: Automatically

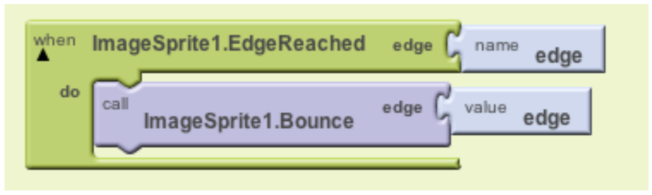
Introduce sprite properties such as **Heading**, **Speed**, **X**, **Y**, and **Interval** that are required for automated motion of the sprite, and show students how to set these properties from the Properties panel in the Designer view. Create a simple app where the ball bounces around the screen, using the 'EdgeDetected' event handler and the 'Bounce' procedure in the blocks editor.





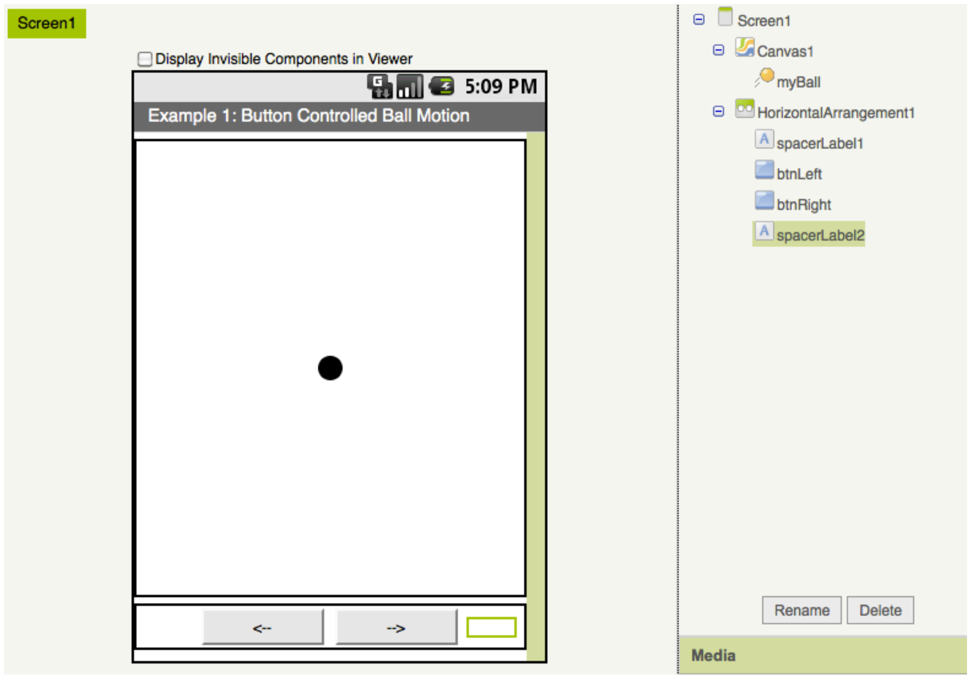
Then, ask the students (individually) to replace the ball with an **ImageSprite**, using an image of their choice (e.g. a fancy ball image from Google), but make it do the same sort of bouncing around using 'edge detected'.




when ImageSprite1.EdgeReached edge
name edge
do
call ImageSprite1.Bounce edge
value edge

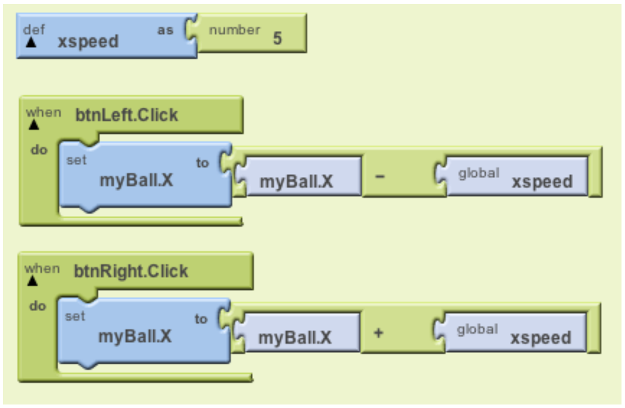
Motion Method 2: Button-controlled ball motion:

Ask students to create an AppInventor project with a single ball on a canvas and left/right buttons below the canvas, as shown below:



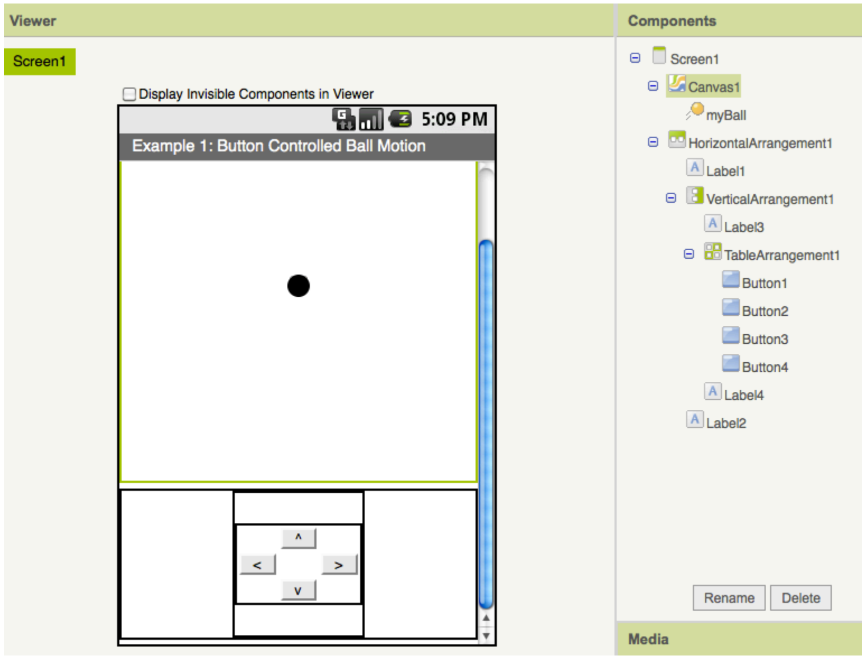
In order to make the ball move, we will update the ball's x position as the left and right buttons are pressed. The amount of pixels that we move the ball *each* time the buttons are clicked is known as the ball's *speed* (in the x direction).

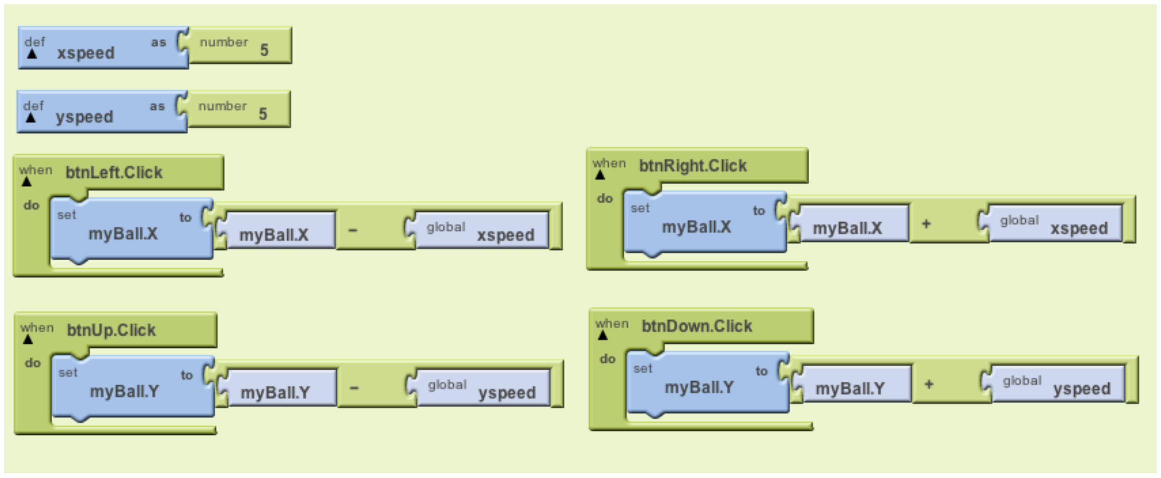
Introduce the concept of **variables**, and discuss how to define your own, and how to access and set their value. Then, ask students to create a variable representing the speed of the ball in the x direction. The blocks code for making the ball move then looks as follows:



```
def xspeed as number 5

when btnLeft.Click
do  set myBall.X to myBall.X - global xspeed

when btnRight.Click
do  set myBall.X to myBall.X + global xspeed
```

Ask students to *extend* this app for up and down motion controlled by up and down arrows:





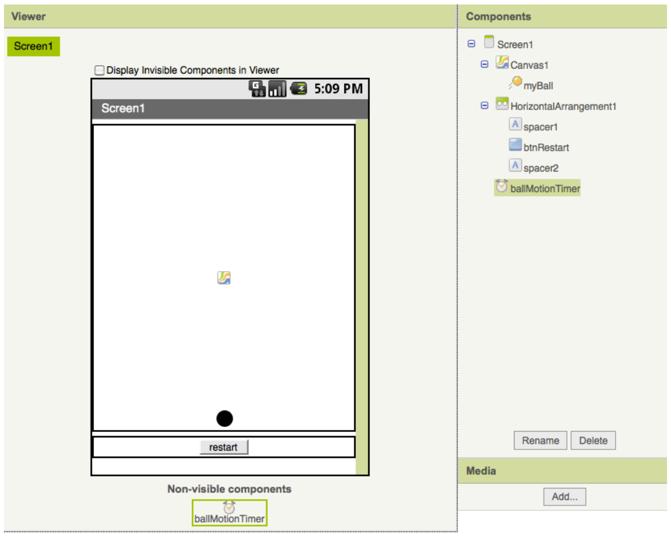
```
def xspeed as number 5
def yspeed as number 5

when btnLeft.Click
do set myBall.X to myBall.X - global xspeed

when btnRight.Click
do set myBall.X to myBall.X + global xspeed

when btnUp.Click
do set myBall.Y to myBall.Y - global yspeed

when btnDown.Click
do set myBall.Y to myBall.Y + global yspeed
```

Note: Students may find it annoying and tiresome to have to keep pressing the buttons repeatedly to make the object move around the screen. There is a way to make the objects move *smoothly* using buttons, but this approach requires knowledge of the Clock component, and will therefore by taught below under "Motion Method 3".

Motion Method 3: Using the AppInventor Clock component

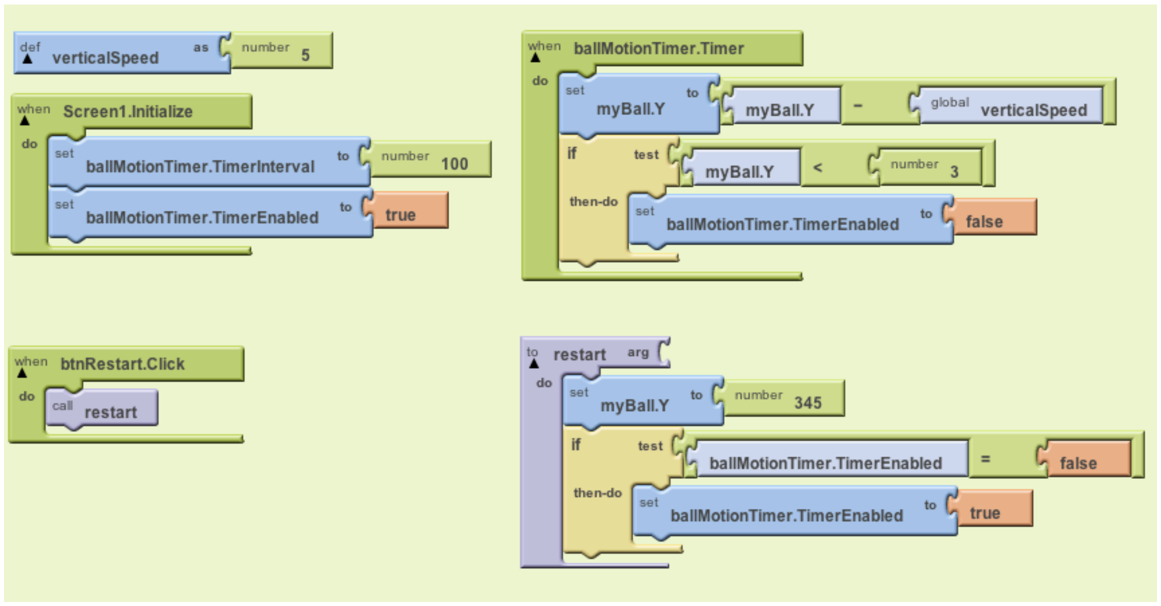
Introduce the **Clock** component, which is a timer that fires every few milliseconds (as specified by the **TimerInterval** property, which can be set either in the designer view or in the blocks editor).

Ask students to create a new project with a Ball, Canvas, and Clock (which will show up below the screen as a **non-visible component**).



We now make the ball move up a few pixels (as determined by the value of the speed variable) every time the timer fires. Once the ball reaches the top of the screen, we ask the timer to stop firing, and we wait for the user to click the 'restart' button. Clicking the restart button causes the ball to reset its location to the bottom of the screen and turns on the timer once again.

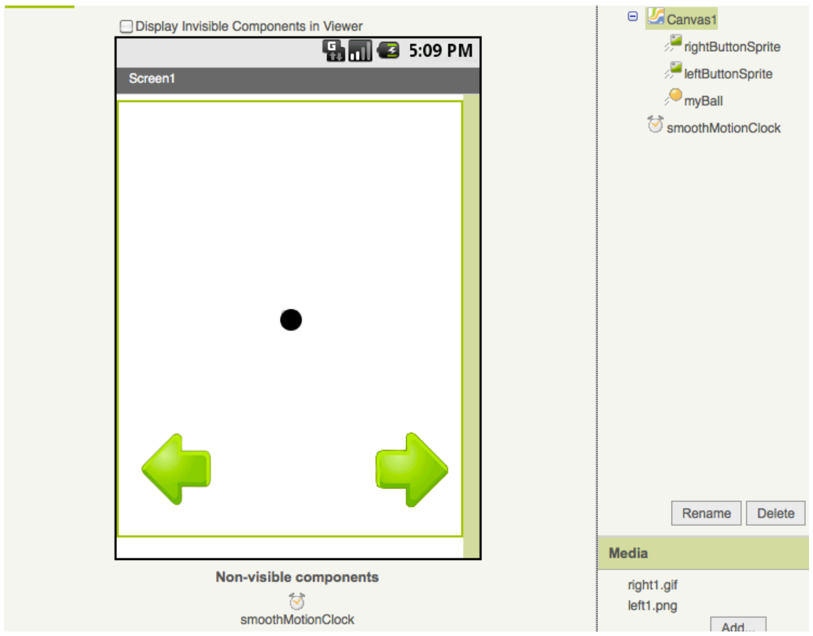
We use this opportunity to teach how to define and call your own **procedure**, by turning the 'restart' action into its own procedure. The blocks code is shown below:



```
def verticalSpeed as number 5

when Screen1.Initialize
do  set ballMotionTimer.TimerInterval to number 100
    set ballMotionTimer.TimerEnabled to true

when ballMotionTimer.Timer
do  set myBall.Y to myBall.Y - global verticalSpeed
    if test myBall.Y < number 3
    then-do set ballMotionTimer.TimerEnabled to false

when btnRestart.Click
do  call restart

to restart arg
do  set myBall.Y to number 345
    if test ballMotionTimer.TimerEnabled = false
    then-do set ballMotionTimer.TimerEnabled to true
```

Extra Credit: It is annoying to make the ball move using buttons, since the user has to press the buttons repeatedly to keep the ball moving. He/she cannot simply leave his/her finger on the button and allow the ball to move *while* the button is pressed. This is because the Click handler for Button components only allows detection of taps and gives no information about how long it was pressed. So, in this extra-credit exercise we show how to achieve motion of sprites using ImageSprite objects as buttons and Clock components to ensure smooth motion.

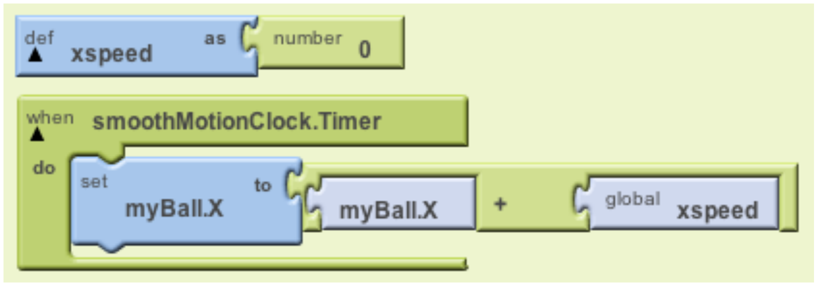
Ask the student to design the following interface, with a ball, two ImageSprites representing left and right buttons, and a Clock:



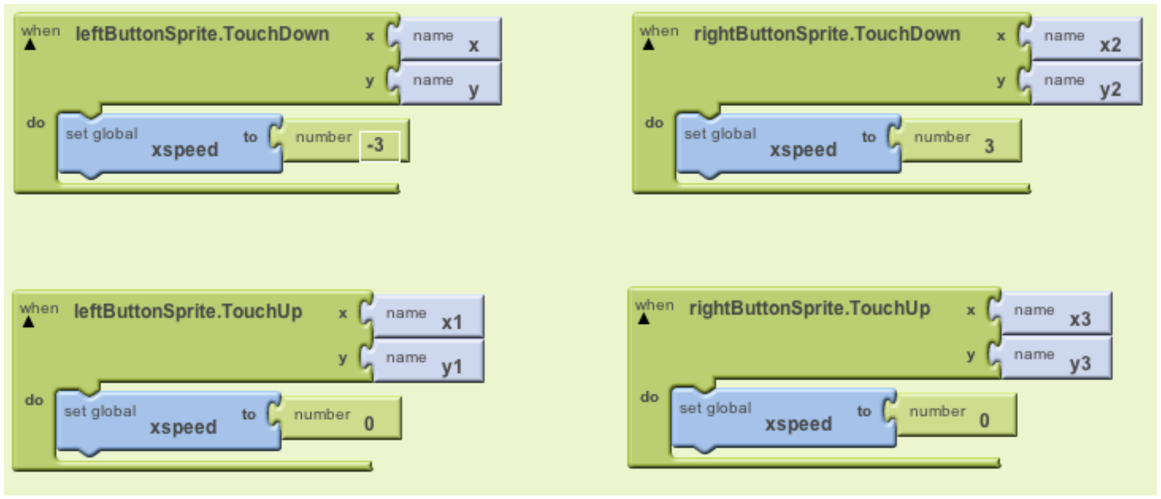
ImageSprite objects, like Balls, have a number of event-handler methods such as "Touched", "TouchDown", and "TouchUp". A TouchDown event is fired whenever a user's finger touches down on an image sprite, and a TouchUp is fired when the finger lifts off that image sprite. The Touched is fired only if the TouchDown and TouchUp occur quickly in succession. The Click event-handler works the same way that the Touched event-handler works for sprites, which is why, for this exercise, we will use TouchDown and TouchUp since they provide more information about when the finger was put down and when it was lifted.

The students' goal in this exercise is to build an app where the ball moves left / right while the user presses the left and right button and it stops moving as soon as the player lets go of the left / right buttons.

As before, we create a variable for the ball's speed, setting it to zero initially. In the clock's Timer event handler, we update the ball's horizontal position by its speed.


```
def xspeed as number 0
when smoothMotionClock.Timer
do  set myBall.X to  myBall.X + global xspeed
```

The magic occurs in the TouchUp / TouchDown event handling:



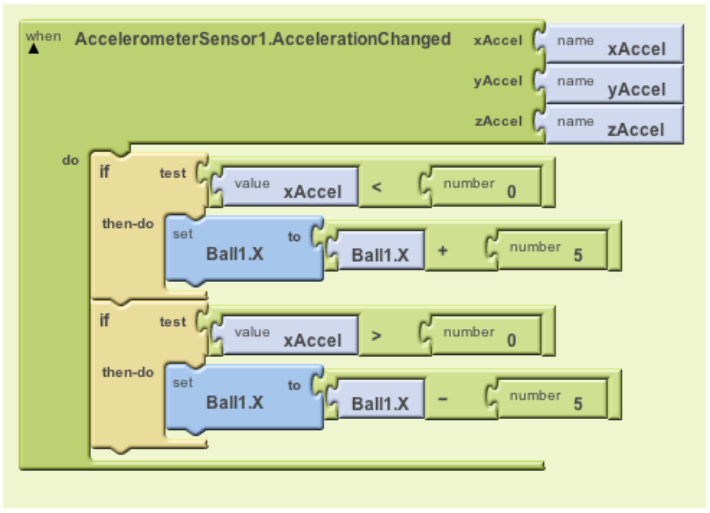
```
when leftButtonSprite.TouchDown  x name x  y name y
do set global xspeed to number -3

when rightButtonSprite.TouchDown  x name x2  y name y2
do set global xspeed to number 3

when leftButtonSprite.TouchUp  x name x1  y name y1
do set global xspeed to number 0

when rightButtonSprite.TouchUp  x name x3  y name y3
do set global xspeed to number 0
```

While the user is pressing the left-button, the ball's speed remains -3, but as soon as the left button is un-pressed, the ball's speed is set back to zero. Likewise, the ball's speed is +3 *while* the right button is pressed but becomes zero as soon as the finger is lifted.

The result is smooth motion of the ball using buttons (which are actually ImageSprite components) and a Clock.

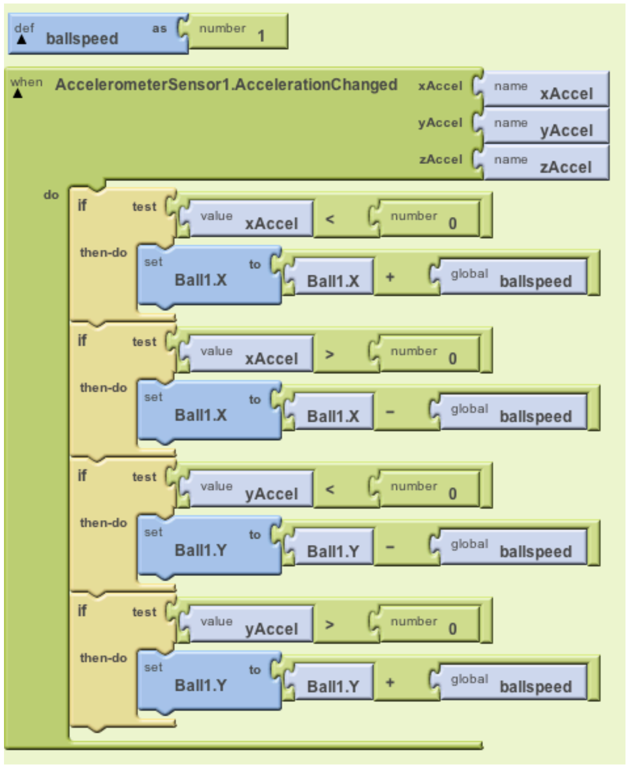
Motion Method 4: Using the Accelerometer

The Accelerometer is a built-in sensor in Android devices that senses changes in acceleration. Here, we use the accelerometer to make the ball move left and right as the phone is tilted left and right, hence avoiding the need for buttons and other forms of tactile input.



```
when AccelerometerSensor1.AccelerationChanged  xAccel  name xAccel
                                               yAccel  name yAccel
                                               zAccel  name zAccel
do  if test  value xAccel  <  number 0
       then-do  set Ball1.X to  Ball1.X  +  number 5
    if test  value xAccel  >  number 0
       then-do  set Ball1.X to  Ball1.X  -  number 5
```

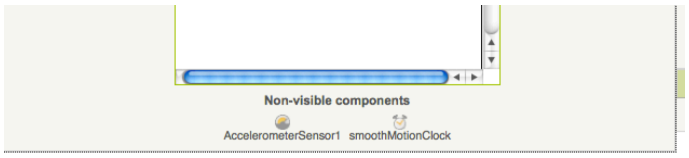
If students finish early, ask them to extend the program to also detect acceleration changes in the y direction (to move the ball up and down):



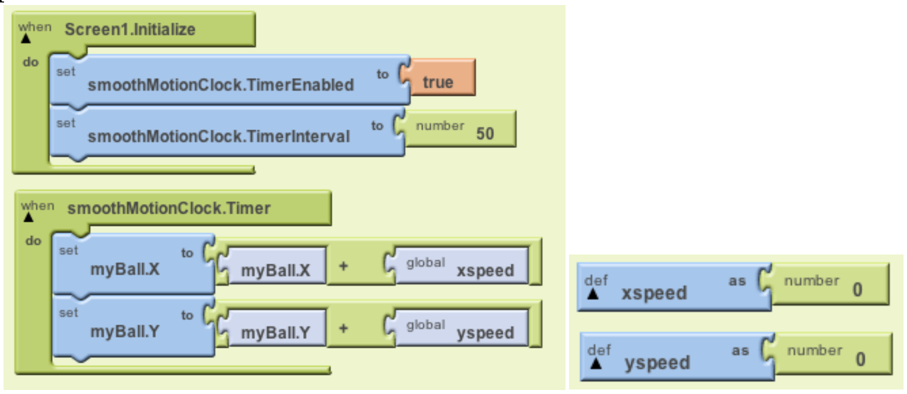
```
def ballspeed as number 1

when AccelerometerSensor1.AccelerationChanged  xAccel  name xAccel
                                               yAccel  name yAccel
                                               zAccel  name zAccel
do  if test  value xAccel  <  number 0
       then-do  set Ball1.X to  Ball1.X  +  global ballspeed
    if test  value xAccel  >  number 0
       then-do  set Ball1.X to  Ball1.X  -  global ballspeed
    if test  value yAccel  <  number 0
       then-do  set Ball1.Y to  Ball1.Y  -  global ballspeed
    if test  value yAccel  >  number 0
       then-do  set Ball1.Y to  Ball1.Y  +  global ballspeed
```

Extra Credit: Students will note that while the above code works, the motion of the ball is not smooth. In order to achieve smooth motion, students should use the **Clock** component *in conjunction* with the **Accelerometer** sensor, following the steps below.

Smooth Motion Step 1: Add a Clock component to the project. In the below screenshots, the clock is called "smoothMotionClock".

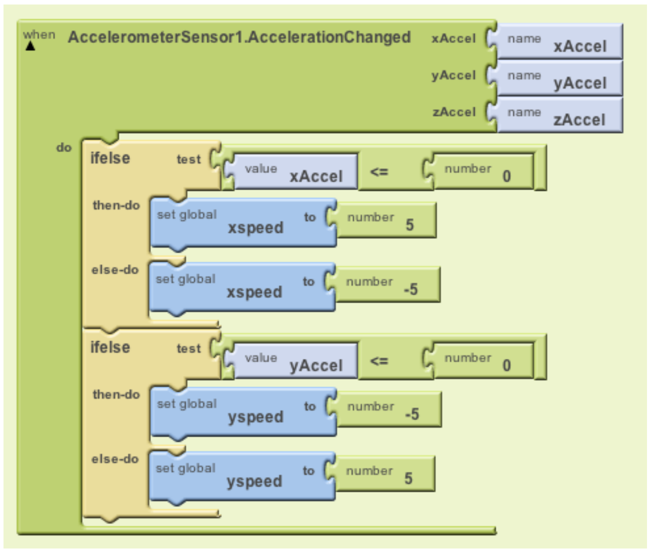


Smooth Motion Step 2: In the blocks code, create variables for speed in the x and y direction, initialize the clock's timer interval in the Screen's Initialize event, and handle the clock's Timer event by updating the ball's x and y location by the appropriate speed variables.



```
when Screen1.Initialize
do  set smoothMotionClock.TimerEnabled to true
    set smoothMotionClock.TimerInterval to number 50

when smoothMotionClock.Timer
do  set myBall.X to myBall.X + global xspeed
    set myBall.Y to myBall.Y + global yspeed

def xspeed as number 0

def yspeed as number 0
```

Smooth Motion Step 3: In the AccelerationChanged event handler, instead of updating the ball's position as you did in the previous version of this app, simply set the value of the x and y speed variables. The ball's position will be updated by the Clock timer and therefore will be unaffected by factors such as how quickly the phone is tilted.

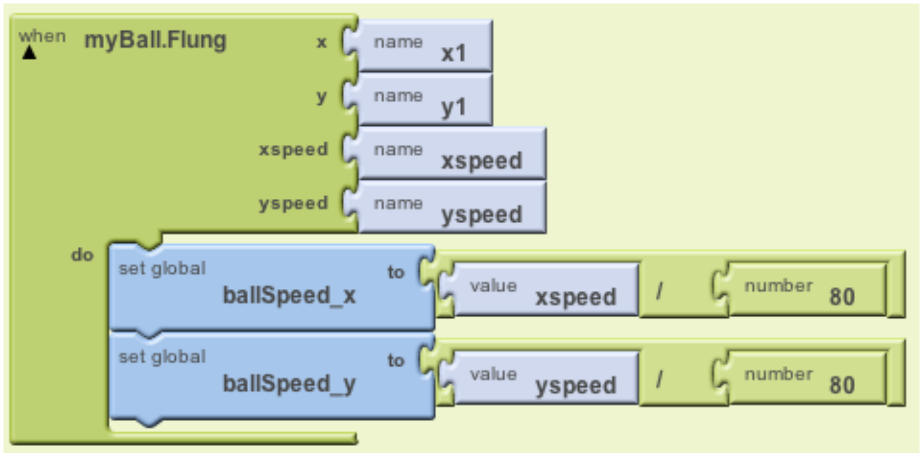


```
when AccelerometerSensor1.AccelerationChanged  xAccel name xAccel
                                               yAccel name yAccel
                                               zAccel name zAccel
do  ifelse   test value xAccel <= number 0
    then-do  set global xspeed to number 5
    else-do  set global xspeed to number -5
    ifelse   test value yAccel <= number 0
    then-do  set global yspeed to number -5
    else-do  set global yspeed to number 5
```

Motion Method 5: Using Fling gesture-detection

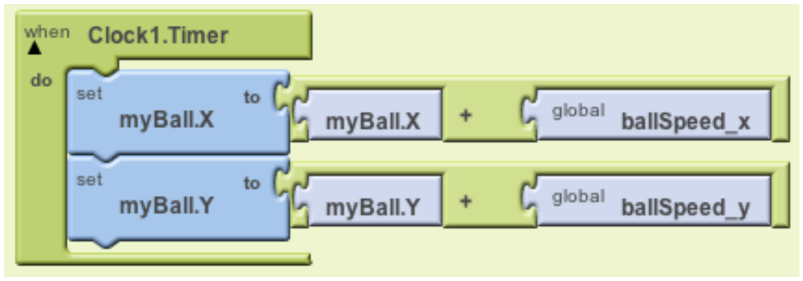
In this section, we teach students how to cause objects to move when flung by the user (i.e. when the user puts a finger down on an object and then quickly swipes in the direction in which it wants the object to move).

If students wish to detect *any* fling gesture that takes place anywhere on the canvas, then students should handle the "Canvas.Flung" event. On the other hand, if they only care about when a particular sprite is flung, then they should handle the Flung event of on that sprite. Below, we show how to detect flings on the "Ball" component that we have been playing around with for the past few exercises.



```
when myBall.Flung
    x       name x1
    y       name y1
    xspeed  name xspeed
    yspeed  name yspeed
do
    set global ballSpeed_x to  value xspeed / number 80
    set global ballSpeed_y to  value yspeed / number 80
```

In the Flung event-handler, we merely set the speed of the ball in proportion to the speed and strength of the fling. That is, the quicker the user moves his/her finger to fling our ball, the quicker our ball should move. The actual updating of the ball's position happens in the "Timer" handler of the Clock:

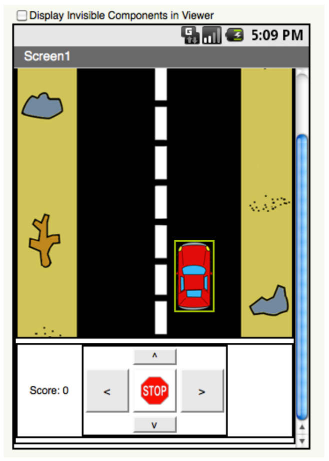


```
when Clock1.Timer
do
    set myBall.X to  myBall.X + global ballSpeed_x
    set myBall.Y to  myBall.Y + global ballSpeed_y
```

Day 3:

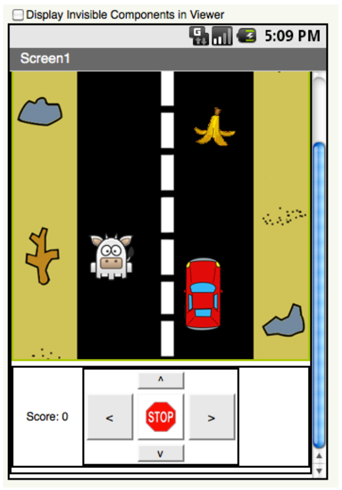
Project 5: Car on a Road

Review previous day's material -- Canvas, Drawing, and motion of sprites -- by building a simple game where a car (represented by an ImageSprite) moves left and right on a road (a Canvas with an image as a background):

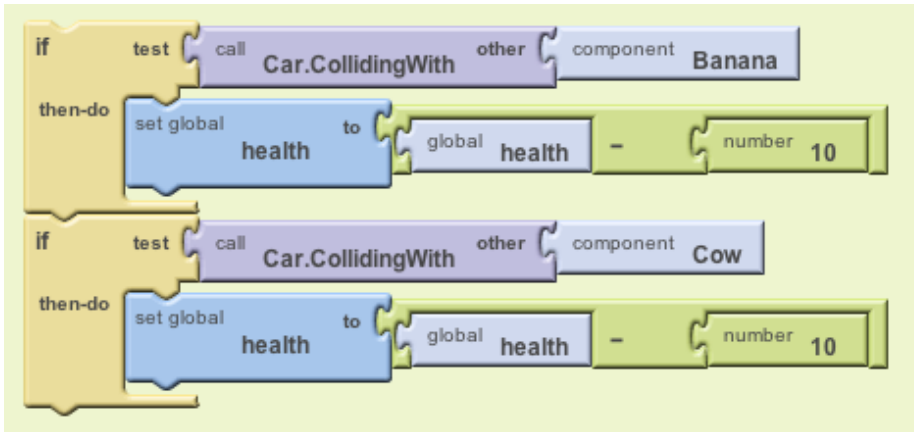


The students may implement the motion of the car using either the Clock component, buttons, or the accelerometer.

Using this same project as a starting point, introduce students to **collision detection**, a very useful facet of game development. Ask them to add one or more obstacles (ImageSprite objects) onto the road, e.g. the banana and the cow in the following image:

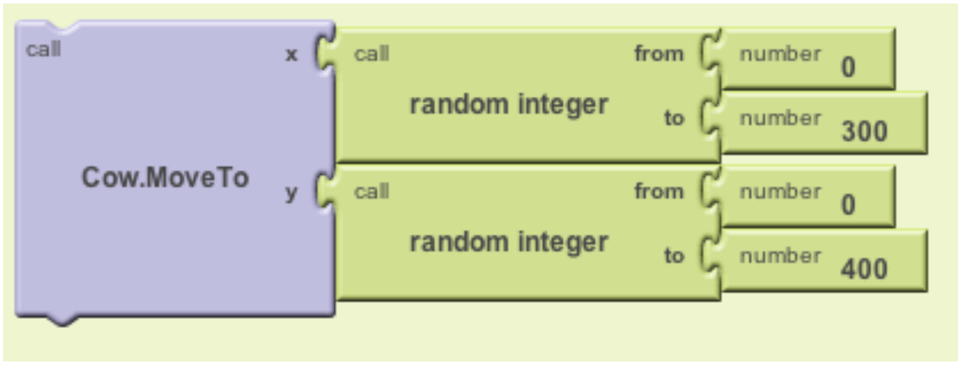


In the blocks code, ask students to insert if-statements (right after the code that updates the car's position) to check if the car is colliding with other obstacles:



Next, introduce students to **random-number generation** by showing them how to make the obstacles on the road appear in random (x,y) coordinate locations using the *random integer* procedure to pick random values for x and y.

In the below blocks code, the Cow's x value is picked randomly from within the range [0, 300] and the y value is picked randomly from the range [0, 400]:

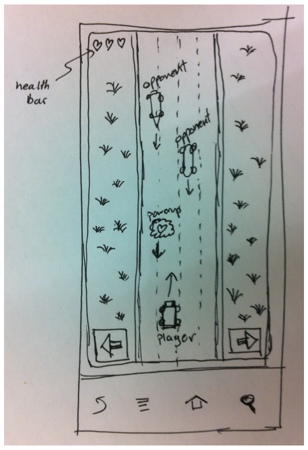


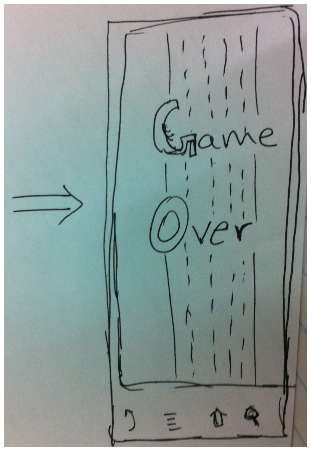
Extra Credit: If students finish early, have them attempt to complete this car obstacle-avoidance game by animating the obstacles such that they appear at the top of the screen, move from top to bottom, and then reset to the top of the screen (at a random x location) and come down again. The students can accomplish this either manually using Clock components or automatically using the built-in ImageSprite properties of Interval, Speed, and Heading (the heading would be 270 degrees if the objects are moving straight down). Note: If multiple obstacles are to be moved simultaneously but at different rates, one Clock component might be necessary for each obstacle.

Project 6: Paper Prototyping

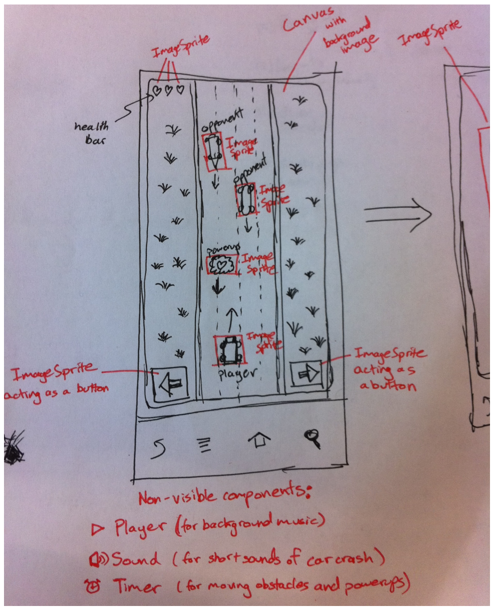
Ask students to brainstorm ideas for games they wish to create by the end of the course (they should be shown examples of existing AppInventor / Android games in order to get their creative juices flowing). Then take them through the steps below.

Step 1: Draw, with pencil or pen, a prototype of the game you wish to create. If the game will have multiple screens, draw out each of those screens.

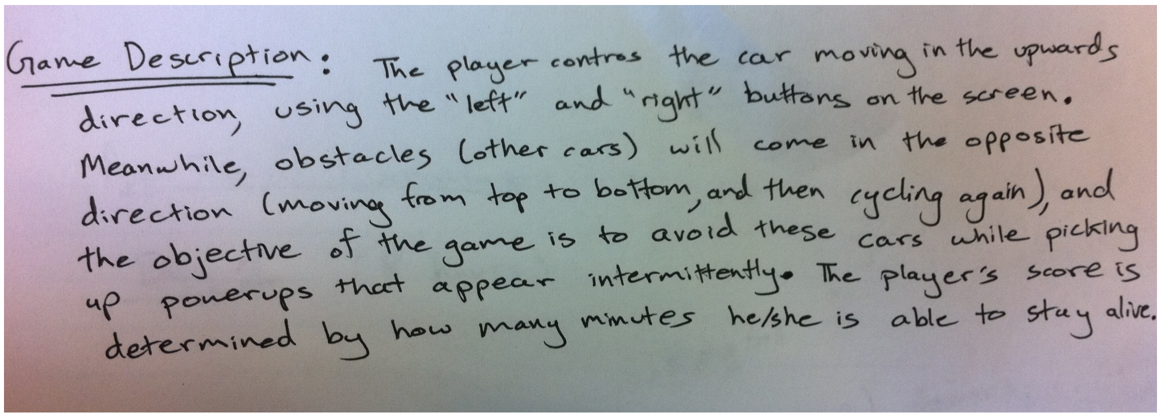




Step 2: On top of the drawings for each screen, write down the names of the AppInventor components you wish to use for each object in the game.


ImageSprite
Canvas with background image
ImageSprite
health bar
opponent
Image Sprite
opponent
Image Sprite
powerup
Image Sprite
Image Sprite
player
ImageSprite acting as a button
ImageSprite acting as a button
Non-visible components:
Player (for background music)
Sound (for short sounds of car crash)
Timer (for moving obstacles and powerups)

<u>Step 3</u>: Write a brief description of the game on a piece of paper. Make the description read-able and easily understandable, you will be sharing it with your peers.



Game Description: The player contros the car moving in the upwards direction, using the "left" and "right" buttons on the screen. Meanwhile, obstacles (other cars) will come in the opposite direction (moving from top to bottom, and then cycling again), and the objective of the game is to avoid these cars while picking up powerups that appear intermittently. The player's score is determined by how many minutes he/she is able to stay alive.

<u>Step 4</u>: Prototype-sharing and Feedback session. Each group or individual should leave their paper prototypes and written description of the game on their table and rotate one spot to the left, where the group can evaluate the neighbor's game. The group should write down any feedback or suggestions they have on the piece of paper that contains the Game Prototype / Description, and then all groups should rotate another spot to the left. This continues until each group has rotated through all tables and provided feedback to every other group.

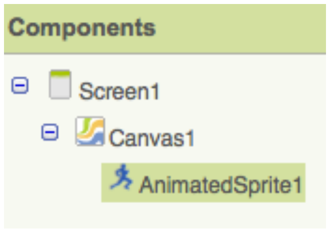
For the rest of the day, students should work on their games in pairs or individually. The Instructor should help groups on a 1-on-1 basis, teaching them topics that they require for their games but haven't yet learned.

Day 4:

Project 7: Fun with Animations

Begin the day by introducing the **AnimatedSprite** component, which may prove very useful to the students in their games.

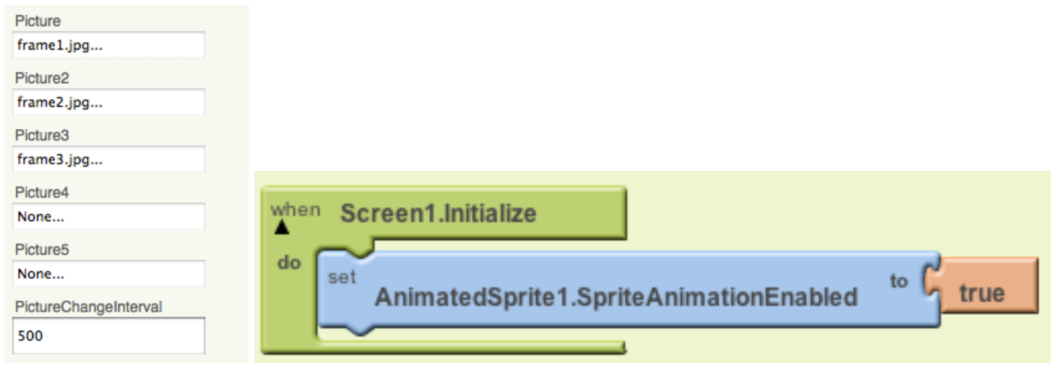
First, students should drag an AnimatedSprite component onto their screen.



Then, they should upload up to 5 images for the animated sprite component using the Properties panel. These images should preferably be multiple frames of the same animation, for example this one:

It is ok if they upload less than 5 pictures; the AnimatedSprite object will automatically flip between the pictures that are uploaded. The students should also set the PictureChangeInterval property to the amount of milliseconds they'd like to wait between each picture change in the animation.

As a last step, students should set the "SpriteAnimationEnabled" property to "true" in their blocks code to start animating their sprite.



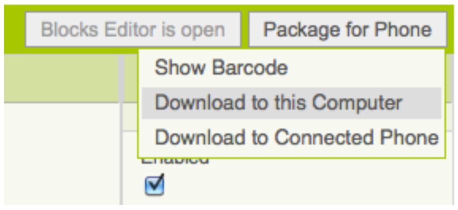
For the rest of the day, the students should work on their games as individuals or pairs. Instructor should assist groups on a 1-on-1 basis, providing help where necessary and teaching new topics that haven't yet been covered to the different groups.

Day 5:

Because this is the final day of the course, students will (hopefully) be extremely excited, energetic, and eager to work on their final projects. If there are any specific topics that they should learn before the end of the course, the morning would be the right time since by the evening the students will be busy making last-minute changes to their games.

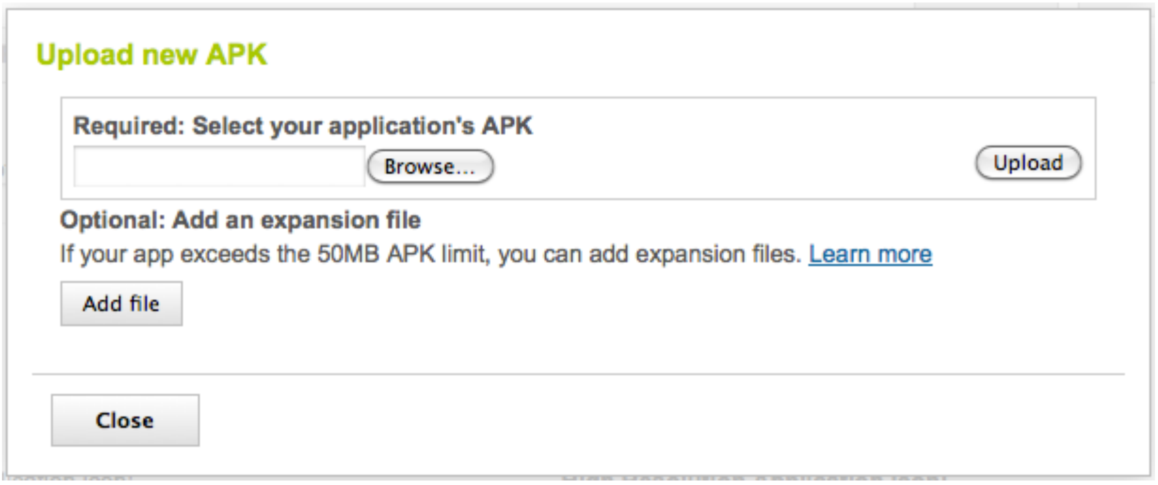
So, before asking the students to begin working on their projects for the day, show them how to **publish apps to Google Play** (formerly known as Android Market) so that the general public can download their apps from the internet to their phone. The steps are outlined below.

Step 1: Click "Download to this Computer" from the designer view of the AppInventor site in order to generate an .apk file for your AppInventor project.



Step 2: Then go to https://play.google.com/apps/publish/Home. If you are not registered as a Google Play developer, you will need to pay a registration fee, create a developer profile, and accept some agreements. After you've done that (or if you're already a registered developer) you should see the Google Play Developer Console. From this console, click the "Upload Application" button to being the process.

Step 3: Upload your .apk file when prompted.



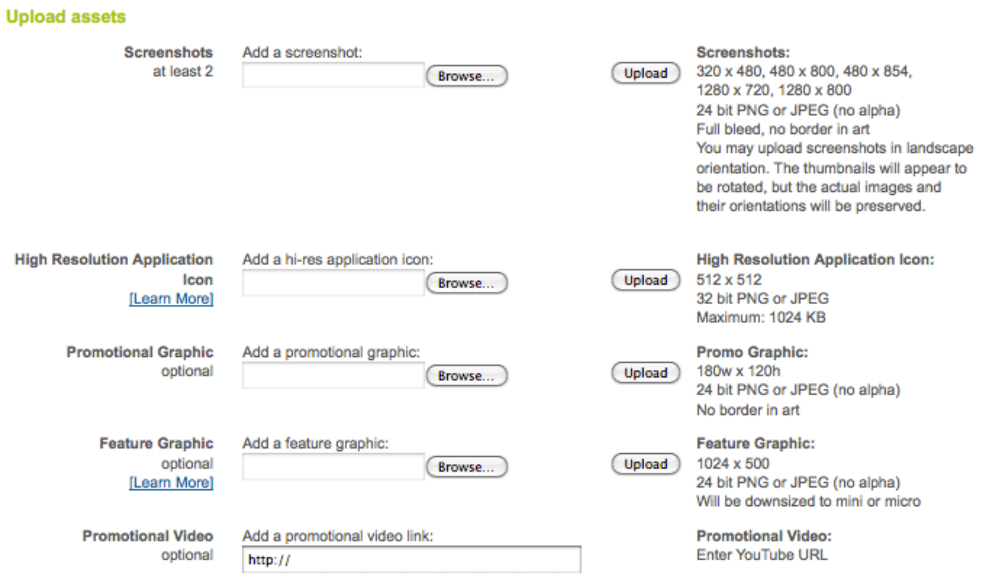
Upload new APK

Required: Select your application's APK

Optional: Add an expansion file

If your app exceeds the 50MB APK limit, you can add expansion files. Learn more

Step 4: Provide other necessary information, e.g. the image icon to be used for your Android application, and the screenshots that should show up when an Android user tries to download your application from Google Play.



Upload assets

Screenshots at least 2	Add a screenshot:	Screenshots: 320 x 480, 480 x 800, 480 x 854, 1280 x 720, 1280 x 800 24 bit PNG or JPEG (no alpha) Full bleed, no border in art You may upload screenshots in landscape orientation. The thumbnails will appear to be rotated, but the actual images and their orientations will be preserved.
High Resolution Application Icon [Learn More]	Add a hi-res application icon:	High Resolution Application Icon: 512 x 512 32 bit PNG or JPEG Maximum: 1024 KB
Promotional Graphic optional	Add a promotional graphic:	Promo Graphic: 180w x 120h 24 bit PNG or JPEG (no alpha) No border in art
Feature Graphic optional [Learn More]	Add a feature graphic:	Feature Graphic: 1024 x 500 24 bit PNG or JPEG (no alpha) Will be downsized to mini or micro
Promotional Video optional	Add a promotional video link: http://	Promotional Video: Enter YouTube URL

Step 5: Click "Publish".

For the rest of the day, the students should work on finishing the games they had started developing the previous day, and the instructor should work with them on a one-on-one basis.

During the final hour of the course, the students should present their apps to each other.

Chapter 6

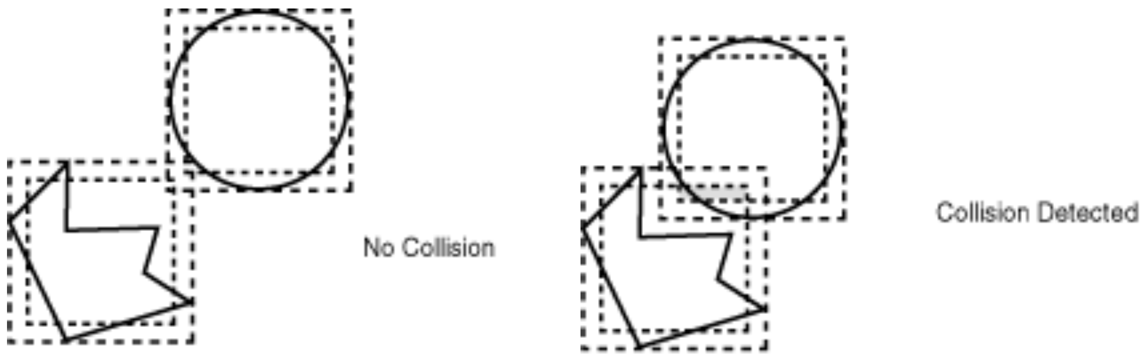
Future Plans and Suggestions

Suggested feature additions.

Chapter 3 identified limitations of AppInventor with respect to game development capabilities, and Chapter 4 described a number of AppInventor modifications that address them. However, there still remain ample opportunities for improvement in AppInventor -- new components can be added and existing ones can be modified so that AppInventor users may more easily develop aesthetically pleasing and professional-looking games. Below is a list of App Inventor modifications that the author believes would be *most* useful to add in the near future, in addition to the modifications he has already made.

Collision Detection

Currently, collisions in AppInventor are detected by checking for an intersection between the bounding rectangles of each ImageSprite. If the two bounding rectangles collide, even if the two objects they encompass don't actually collide, a collision will be detected, and this result would be inaccurate.



It is not hard to see from the figure above, that any game that depends on collision detection between non-rectangular objects would soon be characterized as 'unfair' or 'annoying' due to the errors the collision detection algorithm would make at the corners of the two objects.

It is difficult as well as computationally intensive to do pixel-by-pixel collision detection for all objects in motion in a game. Instead, it is better to take an intermediate approach that performs polygon-based collision detection. In this approach, the game developer would have the option of specifying the shape and size of a custom 'bounding polygon' (triangle, square, pentagon, hexagon, etc.), and then this custom polygon would be used internally by AppInventor to perform collision checks between objects that are in each others' vicinities.

The Android SDK does not provide a collision detection library or algorithm for polygon bounding boxes, unfortunately, so this would need to be implemented manually, using algorithms such as the Separating Axis Theorem.

ShapeSprite

Often in game development, there is a need for simple graphics that can be comprised of shapes such as rectangles, ovals, lines, arcs, and polygons. When such objects are needed, it may be better (in terms of game performance, size of the generated .apk file, and overall look and feel) if the graphics are vector-based instead of images that the developer needed to upload. Furthermore, from a convenience and efficiency point of view, if a game developer wishes to create objects that are simple rectangles, it is annoying to have to create an image of a rectangle and then upload it to AppInventor as an ImageSprite; it would be faster to simply drag a Rectangle sprite onto the project layout.

So, the author proposes the addition of a ShapeSprite, which will be a component that a developer can drag onto his/her project. The developer will select the specific shape of this ShapeSprite from the right-hand-side Properties panel, where a drop-down will be available to allow for selection between Rectangle, Oval, Line, Arc, and Polygon. The chosen polygon shape will then be added to the developer's app as a sprite object that is capable of moving within the canvas, just like a Ball or ImageSprite.

ScrollingBackground

Game developers often rely on scrolling backgrounds to provide the illusion of motion and progress in a game, yet in AppInventor, there is currently no support for implementing this. A Canvas may take a fixed background image, but there are no available properties for determining the *offset* of that backgrounds image, so it is not possible to make that image scroll.

This can be accomplished in multiple ways in Android. One is to use translations on the entire view -- this is convenient if all you're trying to do is a back-andforth translation motion for the background, but if the goal is to loop the background (which is a bitmap resource being drawn onto a canvas), then the right way to implement this would be to keep decrementing the *offset* of the background (so that it moves right-to-left) until it starts to go off screen, and then to start drawing the same background image to fill in the gap left by the first background image that has started going off-screen. So, the AppInventor developer would only need to upload one background image, and specify two additional properties: he/she would need to set the ScrollBackground property to true, and set BackgroundScrollSpeed to a nonzero number representing the number of pixels to scroll the background each interval. The direction of the background's scrolling could then be set by choosing the sign of the scroll speed to be negative or positive.

Below is an example of how to implement scrolling backgrounds using the Android SDK:

```
// decrement the background offset
mBGFarMoveX = mBGFarMoveX - 1;

// calculate the wrap factor for matching image draw
int newFarX = mBackgroundImageFar.getWidth() - (-mBGFarMoveX);

// if we have scrolled all the way, reset to start
if (newFarX <= 0) {
      mBGFarMoveX = 0;
      // only need one draw
      canvas.drawBitmap(mBackgroundImageFar, mBGFarMoveX, 0, null);
} else {
        // need to draw original and wrap
        canvas.drawBitmap(mBackgroundImageFar, mBGFarMoveX, 0, null);
        canvas.drawBitmap(mBackgroundImageFar, newFarX, 0, null);
}
```

Future plans and experiments.

Now that the author of this thesis has made modifications to AppInventor and refinements to his Game Development curriculum, it is important to test whether these changes actually enable students to make better-looking games or to pick up game development concepts more quickly.

To this end, the author will be performing the following two experiments in the near future:

Teaching in India:

In June - July 2012, the author will be personally conducting some AppInventor game development courses for high-school students in Delhi and Bangalore using his refined curriculum and the modified version of AppInventor. This will be an excellent way to validate and confirm whether the modifications and refinements make any impact on the students' learning and on the quality of their games. Each course will be 40-hours long and will be taught over 1 week (8 hrs / day).

Teaching in China:

The author will be traveling to China with a group of other MIT students in August 2012. There, he will lead the instruction of an AppInventor-based Entrepreneurship course, the goal of which would be to empower students to build Android apps that address problems in the students' community. Although the focus of this course will not be on Game Development, the author will use game development to initially get the students excited about using AppInventor and building android apps.

It will be interesting to see if the flashcards developed by the author, as well as the Media Library, aid students in learning AppInventor more quickly and designing aesthetically pleasing apps more efficiently.

While teaching these courses in June - August 2011, the author will continue to work with the Center for Mobile Learning at MIT to ensure that other students familiar with AppInventor are also traveling to communities around the world to teach AppInventor, making use of the resources developed by the author as a part of this thesis (e.g. the curriculum, flashcards, and media library).

Chapter 7

Contributions and References

Contributions

- Identified and articulated the relevance and importance of Game Development to educational tools such as AppInventor.
- Identified existing and missing features for Game Development in AppInventor based on first-hand teaching experiences in India and Boston.
- Developed two iterations of a hands-on curriculum for a 40-hour 1-week AppInventor course on Game Development.
- Compiled a Media Library, consisting of images and audio, to be provided to students attempting to build games with AppInventor.
- Created a set of 20 Flash Cards of AppInventor game development fundamentals, to be used by anybody learning AppInventor.
- Added AnimatedSprite and ProximitySensor components and implemented modifications to Canvas and sprite components enabling more precise touch-handling and Fling gesture detection.

References

[1] MIT AppInventor. Building Your First App (Hello Purr).
<http://beta.appinventor.mit.edu/learn/setup/hellopurr/hellopurrphonepart1.html>

[2] MIT AppInventor. PaintPot Tutorial (Part 1).
<http://beta.appinventor.mit.edu/learn/tutorials/paintpot/paintpot-part1.html>

[3] Android Developers Website. Scrolling Backgrounds Example: JetBoy Game.
<http://developer.android.com/resources/samples/JetBoy/src/com/example/android/jetboy/JetBoyView.html>

[4] Separating Axis Theorem. Detecting Polygon Collisions.
<http://www.codeproject.com/KB/GDI-plus/PolygonCollision.aspx>