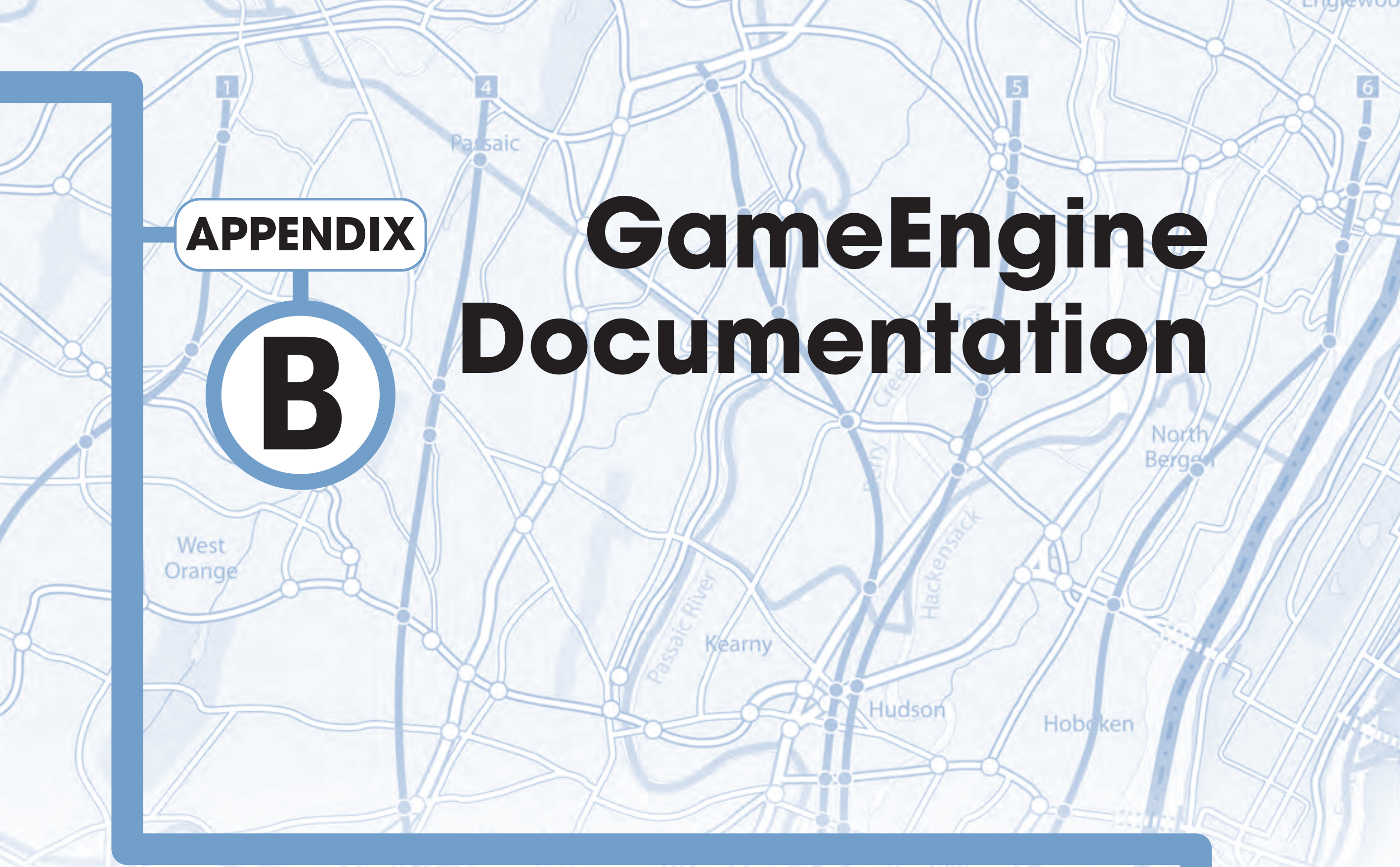

APPENDIX B

GameEngine Documentation

The gameEngine module is powerful and fun to use. Even if you haven't read all the way through the book, you can use this module to create some interesting and fun games.

Installing the Module

The easiest way to use gameEngine is to simply copy the gameEngine file into your program's working directory. You can then import gameEngine to have access to all the features of the module. If you want to distribute a game using gameEngine, just include gameEngine.py.

Transfer

Please check Appendix C for more information on packaging and distributing your modules and games.

If you really like gameEngine, you can install it as a module, and then it will be available to all your Python programs, just like pygame.

Here's how you install gameEngine as a module:

1. **Extract the files in gameEngine-1.1.zip to a temporary directory.**
2. **Open a command shell and move to the directory.**

   For example, use a terminal shell in Linux/Mac or a DOS shell in Windows. Move to the temporary directory.
3. **Install the program.**

   You'll find a Python program in the directory called setup.py. Run that program with the install directive, like this:

   ```
   python setup.py install
   ```

This program will automatically copy gameEngine.py to an appropriate spot in your Python system so it will be available to all your Python programs.

Transfer

The setup.py program and installation process are described in Appendix C.

Using the Scene Class

The gameEngine Scene class is a simple but powerful tool that encapsulates the IDEA/ALTER framework. You can make an instance of the Scene class right away, or you can extend the Scene class to make a custom starting point for your games.

Once you've created a Scene, you can add sprites to its sprites attribute. You can also modify the background object, and even make new sprite lists that will all automatically be updated at the right time. You can add any sprites you wish to a scene, regular sprites, instances of the SuperSprite class, gameEngine widgets, or extensions of any of these things.

When you want the game to begin, call your scene instance's start() method to get things rolling. Of course you can stop a scene with the stop() method. One program can have multiple scenes to handle various states, sub-games, instruction screens, or whatever you want.

If you've created an extension of the Scene class, you can override one or both of two event-handling functions. The doEvents() method passes a copy of the event object from the main loop so you can do any traditional event-handling there. The update() method runs every frame (just like pygame.sprite.Sprite's update() method) so you can use it to put any code you wish.

Table B-1 describes the Scene attributes, which can be modified externally.

Table B-1 Public Scene Attributes

Name	Type	Description
`background`	Surface	The background drawing surface of the scene.
`screen`	Surface	The primary drawing surface of the scene.
`sprites`	List of sprite objects	The primary sprite group. You can create other sprite groups with the `Scene` class's `makeSpriteGroup()` and `addGroup()` methods.

Table B-2 shows the publicly accessible methods of the `Scene` object.

Table B-2 Public Scene Methods

Name	Parameters	Description
`__init__()`	<none>	Initiator. If you're writing an extension of the `Scene` class, be sure to call `Scene.__init__(self)` at the beginning of the method.
`start()`	<none>	Starts the `__mainLoop()` method.
`stop()`	<none>	Ends the `Scene` object's main loop and sends control back to the calling program.
`makeSpriteGroup (sprites)`	`sprites`: a list of sprites to add to the new sprite group.	Returns a sprite group.
`addSpriteGroup (group)`	`group`: a sprite group created through the `Scene` class's `makeSpriteGroup()` method or the `pygame.sprite.Group()`	Adds the group to the `Scene`'s list of groups. Group is automatically cleared, updated, and redrawn inside main loop.
`setCaption (title)`	`title`: a string containing new caption	Sets the window caption to the text specified by `title`.

The Scene methods shown in Table B-3 are meant to be overridden in subclasses of Scene. Both are empty in the standard Scene class. They are used to add event or collision functionality to extensions of the Scene class.

Table B-3 Scene Override Methods

Name	Parameters	Description
doEvents(event)	event: the event object passed to the doEvents() method.	Override this method to write code that has access to the event object and runs every frame.
update()	<none>	Override this method to write code that runs every frame. Note that doEvents() is called before update().

Using the Label Class

The gameEngine Label class is a lightweight but powerful GUI widget. Use a label instance when you want to easily place text on the screen. All Label control is done through its attributes (see Table B-4). All attributes have a default value, so you don't absolutely have to set any of them (but you should for best results). You can change any Label attributes even after the label has been displayed.

Table B-4 Public Attributes of the Label Class

Name	Type	Description
font	pygame.font.Font instance	The font and size that will be used in this label.
text	string	Text to display.
fgColor	color tuple (R, G, B)	Foreground color of text.
bgColor	color tuple (R, G, B)	Background color behind text.
center	position tuple (x, y)	Position of label center, used to position entire label.
size	size tuple (width, height)	Used to set label's size.

Using the Button Class

The gameEngine Button class is an extension of the Label class that has the added ability to detect mouse clicks. It has two Boolean attributes for detecting mouse activity. active is True if the mouse's left button is down over the button. clicked is True if the mouse left button was pressed and released over the button.

The default background color is set to gray to differentiate it from the Label, but you can change it to whatever you wish. Table B-5 lists the attributes you can modify.

Table B-5 Public Attributes of the Button Class

Name	Type	Description
font *	pygame Font class	The font that will be used in this label.
text *	string	Text to display.
fgColor *	color tuple (R, G, B)	Foreground color of text.
bgColor *	color tuple (R, G, B)	Background color behind text.
center *	position tuple (x, y)	Position of label center, used to position entire label.
size *	size tuple (width, height)	Used to set label's size.

* *This attribute is inherited from the Label class.*

The Button class has two Boolean attributes that are not intended to be changed from the outside, as shown in Table B-6. Instead, these attributes report the current status of the button.

Table B-6 Read-Only Attributes of the Button Class

Name	Type	Description
Active	Boolean	True if mouse button is down and mouse is over button rect.
Clicked	Boolean	True if mouse was pressed and released over button rect.

Using the Scroller Class

gameEngine includes a simple scroller that serves the purpose of a scroll bar in traditional GUI environments. The main purpose of the scroller is to act as a graphical way to edit and view numerical input. The Scroller class is an extension of the Button class, so it has all the characteristics of Button (including those inherited from the Label class). Scroller also has its own attributes used to control the scroller's behavior (see Table B-7).

The scroller has a numeric value attribute that ranges from minValue to maxValue. If the user clicks the mouse on the left half of the scroller, the value is reduced by increment. If the user clicks on the right half, the value is increased by increment.

Table B-7 Public Attributes of the Scroller Class

Name	Type	Description
font *	pygame.font.Font object	The font that will be used in this label.
text *	string	Text to display.
fgColor *	color tuple (R, G, B)	Foreground color of text.
bgColor *	color tuple (R, G, B)	Background color behind text.
center *	position tuple (x, y)	Position of label center. Used to position entire label.
size *	size tuple (width, height)	Used to set label's size.
Value	integer or float	The numeric value that the scroller displays.
minValue	integer or float	Minimum value attainable by scroller. Can be negative.
maxValue	integer or float	Maximum value attainable by scroller.
Increment	integer or float	If mouse is currently clicked, change this amount.

* *This attribute is inherited from the Label class.*

Information Kiosk

The scroller also inherits the active and clicked attributes from the Button class, but these are not meant to be used externally.

Introducing the SuperSprite Class

The SuperSprite class is a powerful and flexible extension of the pygame.sprite.Sprite class. SuperSprite incorporates many of the sprite features described throughout this book. For example, you can set a SuperSprite instance's speed and direction, and the sprite will not only rotate automatically to face that direction, but also travel in that direction at the indicated speed. If you prefer, you can indicate the sprite's dx and dy attributes, and it will move accordingly.

The SuperSprite class has five different boundary-checking behaviors built in. It also has built-in functions to speed up, slow down, and rotate. If you want to use complex physics, you can add a force vector to the sprite, and this new vector will be added to the sprite's current motion vector.

SuperSprite has a couple of features designed for collision detection. You can check to see if the sprite collides with a particular sprite or an entire group of sprites. If you need more specific information, you can easily determine the distance between the sprite and any other point, or the angle between the sprite and a point.

A few utility methods round out the SuperSprite class's bag of tricks. It has a method that prints out the sprite's current position and motion vector (useful for debugging). It also has Boolean methods that can determine whether the sprite is currently being dragged or clicked. Finally, you can change the sprite's image attribute easily to make the sprite handle any bitmap.

The SuperSprite object (like the Scene object) can be instantiated as is, or it can be extended to form your own class based on SuperSprite. If you extend SuperSprite, you can add your own event-handling code. Although the SuperSprite is a powerful beast, it's pretty easy to use. The upcoming section offers a rundown of its attributes.

Public attribute of the SuperSprite

The SuperSprite object has some attributes that could conceivably be referred to from outside the method, but generally there isn't a good reason to do so. If possible, use methods to control the behavior of the SuperSprite object.

Step Into the Real World

The one place I tend to use SuperSprite attributes directly is in the dx and dy attributes. If you do the same, remember also to call the updateVector() method to make sure your changes are recorded. (Of course, if you use the built-in vector-changing methods, you won't need to do this.)

Public methods of the SuperSprite

The `SuperSprite` has a *lot* of methods (29 public and 4 private). To manage all these choices, I'll break them into categories.

Whenever you use the `SuperSprite` object, you'll almost always use a few extremely common methods — which are shown in Table B-8.

Table B-8 SuperSprite's Essential Methods

Name	Parameters	Description
`__init__(`*`scene`*`)`	*`scene`*: a `gameEngine Scene` object. The scene to which this instance is associated	Initializes the `SuperSprite` instance. If you are extending the class, be sure to call `SuperSprite.__init__ (self)` at beginning of method.
`setImage(`*`image`*`)`	*`image`*: the filename of an image (If the image has a front, the front of the image should face East.)	Used to set the `image` attribute. Creates an image master that can be automatically rotated as needed.
`setPosition (`*`position`*`)`	*`position`*: (x, y) coordinates of the desired position on-screen	Moves the sprite immediately to the given coordinates.
`setSpeed(`*`speed`*`)`	*`speed`*: The number of pixels that the sprite will travel per frame	Sets the speed to any arbitrary value, including negative values. No speed limits imposed.
`speedUp(`*`amount`*`)`	*`amount`*: the pixel-per-frame increase in speed. Can be floating-point and/or negative value. (Negative will slow the sprite down and eventually cause the sprite to move backward.)	Changes the speed by *`amount`* pixels per frame. Respects speed limits (–3 to +10 by default, or changed by `setSpeedLimits()`).
`setAngle(`*`dir`*`)`	*`dir`*: an angle in degrees (East is 0, increases counterclockwise)	Arbitrarily sets both the visual rotation and the direction of travel to *`dir`*.
`turnBy(`*`amount`*`)`	*`amount`*: an angle in degrees (negative values turn clockwise, positive values turn counterclockwise)	Changes both visual rotation and direction of travel by *`amount`* degrees.

Sometimes you need more specific controls for the sprite's motion. The methods shown in Table B-9 give you several techniques for altering or setting the sprite's motion vector.

Table B-9 SuperSprite's Vector-Manipulation Methods

Name	Parameters	Description
`setDX(`*`dx`*`)`	*`dx`*: new `dx` value	Changes the `dx` attribute of the sprite. Use this method instead of directly assigning a `dx` value.
`setDY(`*`dx`*`)`	*`dy`*: new `dy` value	Changes the `dy` attribute of the sprite. Use this method instead of directly assigning a `dy` value.
`addDX(ddx)`	*`ddx`*: change in `dx`	Changes the `dx` attribute of the sprite by the indicated amount.
`addDY(ddx)`	*`ddy`*: change in `dy`	Changes the `dy` attribute of the sprite by the indicated amount.
`setComponents ((`*`dx, dy`*`))`	*`dx, dy`*: new motion-vector components	Assigns `dx` and `dy` in one step.
`updateVector()`	<*none*>	If you must set `dx` or `dy` attributes directly, this method makes the change permanent when invoked.
`moveBy(`*`vector`*`)`	*`vector`*: motion vector in component form (`dx`, `dy`)	Moves the sprite according to the vector without affecting rotation, direction, or speed.
`addForce(`*`amt, angle`*`)`	*`amt`*: length of force vector *`angle`*: angle of force vector	Adds force vector to current speed and direction of travel. Used for simulating skidding, gravity, and so on.
`forward(`*`amt`*`)`	*`amt`*: number of pixels to move	Moves in the current facing direction. Does not change speed.
`rotateBy(`*`amt`*`)`	*`amt`*: degrees of rotation to add to visual rotation	Changes visual rotation without affecting direction of travel.

Table B-10 lists several methods for adding handy features to the `SuperSprite`.

Table B-10 SuperSprite's Utility Methods

Name	Parameters	Description
`setBoundAction(`*`action`*`)`	*`action`*: one of the following constants: `WRAP` around screen `BOUNCE` off boundary `STOP` at edge of screen `HIDE` offstage and stop `CONTINUE` indefinitely	Sets the boundary action to the specified value. If none is specified, then `WRAP` is the default behavior.
`setSpeedLimits(`*`max, min`*`)`	*`max`*: maximum speed (pixels per frame) *`min`*: minimum speed (pixels per frame)	Sets the upper and lower speed limits. Respected by the `speedUp()` method.
`mouseDown()`	*<none>*	Returns `True` if mouse button is currently pressed over sprite. Can be used for drag-and-drop.
clicked()	*<none>*	Returns `True` if mouse button is pressed and released over sprite. Makes sprite act like a button.
`collidesWith(`*`target`*`)`	*`target`*: any sprite (ordinary or `SuperSprite`)	Returns `True` if colliding with *`target`*.
`collidesGroup(`*`group`*`)`	*`group`*: a sprite group	Returns `True` if a sprite in the group was hit in this frame, or None if there are no collisions between sprite and group.
`distanceTo(`*`point`*`)`	*`point`*: a specific set of (x, y) coordinates on-screen	Determines the distance from the center of the sprite to *`point`*.
`dirTo(`*`point`*`)`	*`point`*: a specific set of (x, y) coordinates on-screen	Determines direction from center of sprite to *`point`*.
`dataTrace()`	*<none>*	Prints `x`, `y`, `speed`, `dir`, *`dx`*, and `dy`. Used for debugging.

SuperSprite's checkEvents() and checkBounds() override methods

The SuperSprite has one method that is meant to be overridden in subclasses of SuperSprite. Like the update() event in Scene, the checkEvents() method is designed to be overridden. Use this method to add event handling to descendants of SuperSprite. If you have a boundary-checking situation that isn't covered by the standard boundary actions, you can also override checkBounds() (see Table B-11).

Table B-11 SuperSprite's Overwrite Methods

Name	Parameters	Description
checkEvents()	*<none>*	Overwrite this method to add code that will run on each frame (usually event- or collision-handling code).
checkBounds()	*<none>*	Overwrite this method if you need a boundary-checking technique that isn't covered in the built-in techniques.

Sample Programs

The following programs illustrate the gameEngine's features in more detail. (For a complete description of these programs, please see Chapter 10.)

The simplest possible gameEngine game

The simpleGE.py program is the gameEngine equivalent to the traditional "Hello World" test program:

```
""" simpleGe.py
    example of simplest possible
    game engine program
"""

import pygame, gameEngine

game = gameEngine.Scene()
game.start()
```

This program uses a standard instance of the Scene class. The default instance has a SuperSprite object built in (which you will usually replace with your own sprites). Even with this extremely limited code, the program runs and the sprite moves around on the screen.

Extending the SuperSprite class

The SuperSprite class is a powerful tool, but if you want to add event-handling capability, you'll need to extend it. The carGE.py program shows how this is done:

```
""" carGE.py
    extend SuperSprite to add keyboard input
"""

import pygame, gameEngine

class Car(gameEngine.SuperSprite):
    def __init__(self, scene):
        gameEngine.SuperSprite.__init__(self, scene)
        self.setImage("car.gif")

    def checkEvents(self):
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            self.turnBy(5)
        if keys[pygame.K_RIGHT]:
            self.turnBy(-5)
        if keys[pygame.K_UP]:
            self.speedUp(.2)
        if keys[pygame.K_DOWN]:
            self.speedUp(-.2)

def main():
    game = gameEngine.Scene()
    game.background.fill((0xCC, 0xCC, 0xCC))

    car = Car(game)
    game.sprites = [car]

    game.start()

if __name__ == "__main__":
    main()
```

The Car class is simply an extension of the SuperSprite class with the checkEvents() method overwritten. To add the custom class, create an instance of the Car class with the scene instance as its argument, add the car to the scene's sprites list, and start up the scene.

Drag-and-drop sprites

You can also customize a sprite to get a version of drag-and-drop behavior, as shown in dragDrop.py:

```
""" dragDrop.py
    illustrate click and mouseDown methods
    with gameEngine
"""

import pygame, gameEngine

class Ball(gameEngine.SuperSprite):
    def __init__(self, scene):
        gameEngine.SuperSprite.__init__(self, scene)
        self.setImage("ball.gif")

    def checkEvents(self):
        #drag and drop
        if self.mouseDown():
            self.setPosition(pygame.mouse.get_pos())

def main():
    game = gameEngine.Scene()
    ball = Ball(game)
    game.sprites = [ball]
    game.start()

if __name__ == "__main__":
    main()
```

The `Ball` class is an extension of `SuperSprite`. I use its `mouseDown()` method to determine whether the mouse pointer is currently over the sprite. If it is, I set the position to the mouse pointer's current position.

Customizing the scene

Sometimes it's more convenient to customize the scene, particularly when you're working with ordinary sprites or the GUI components. The `guiDemoGE.py` program illustrates all the GUI elements. I put them together using an extension of the `Scene` class.

```
""" guiDemoGE.py
    demonstrates the GUI objects in
    gameEngine
"""

import pygame, gameEngine

class Game(gameEngine.Scene):
    def __init__(self):
        gameEngine.Scene.__init__(self)

        self.addLabels()
        self.addButton()
```

```
          self.addScroller()
          self.addMultiLabel()

          self.sprites = [self.lblTitle, self.label,
                          self.lblButton, self.button,
                          self.lblScroller, self.scroller,
                          self.multi]

      def addLabels(self):
          self.lblTitle = gameEngine.Label()
          self.lblTitle.text = "GameEngine GUI Demo"
          self.lblTitle.center = (320, 40)
          self.lblTitle.size = (300, 30)

          self.label = gameEngine.Label()
          self.label.font = pygame.font.Font                       ↩
("goodfoot.ttf", 40)
          self.label.text = "Label"
          self.label.fgColor = (0xCC, 0x00, 0x00)
          self.label.bgColor = (0xCC, 0xCC, 0x00)
          self.label.center = (320, 100)
          self.label.size = (100, 50)

      def addButton(self):
          self.lblButton = gameEngine.Label()
          self.lblButton.center = (200, 180)
          self.lblButton.text = "Button"

          self.button = gameEngine.Button()
          self.button.center = (450, 180)
          self.button.text = "don't click me"

      def addScroller(self):
          self.lblScroller = gameEngine.Label()
          self.lblScroller.text = "scroller"
          self.lblScroller.center = (200, 250)

          self.scroller = gameEngine.Scroller()
          self.scroller.center = (450, 250)
          self.scroller.minValue= 0
          self.scroller.maxValue = 250
          self.scroller.value = 200
          self.scroller.increment = 5

      def addMultiLabel(self):
          self.multi = gameEngine.MultiLabel()
          self.multi.textLines = [
              "This is a multiline text box.",
              "It's useful when you want to",
              "put larger amounts of text",
              "on the screen. Of course, you",
              "can change the colors and font."
              ]
          self.multi.size = (400, 120)
          self.multi.center = (320, 400)

      def update(self):
          if self.button.clicked:
              self.lblButton.text = "Ouch!"

          self.lblScroller.center = (self.scroller.value, ↩
250)

  def main():
      game = Game()
      game.start()

  if __name__ == "__main__":
      main()
```

Using scrollbars

The `rgbScroller.py` program shows an RGB-color-choosing application that uses the `scroller` component.

```
""" rgbScroller.py
    demonstrates use of scrollers
    to make a color picker
"""

import pygame, gameEngine

class ColorScr(gameEngine.Scroller):
    def __init__(self):
        gameEngine.Scroller.__init__(self)
        self.minValue = 0
        self.maxValue = 255
        self.value = 255
        self.increment = 5
        self.format = "<<  %d  >>"

class Game(gameEngine.Scene):
    def __init__(self):
        gameEngine.Scene.__init__(self)

        #make scrollers
        self.scrRed = ColorScr()
```

```
            self.scrRed.center = (320, 200)

            self.scrGreen = ColorScr()
            self.scrGreen.center = (320, 240)

            self.scrBlue = ColorScr()
            self.scrBlue.center = (320, 280)

            self.sprites = [self.scrRed, self.scrGreen,     ↩
self.scrBlue]
            self.setCaption("RGB Scrollers")

        def update(self):
            red = self.scrRed.value
            green = self.scrGreen.value
            blue = self.scrBlue.value
            self.background.fill((red, green, blue))

            #blit the background
            self.screen.blit(self.background, (0, 0))

    def main():
        game = Game()
        game.start()

    if __name__ == "__main__":
        main()
```

This program uses three instances of the `Scroller` class. The GUI objects don't have any event handling built-in, so I extend the `Scene` class and use its `update()` method to handle the event-handling duties.

Creating a full game

A complete game will usually include several custom `Scene` objects. You might have one scene to handle the instructions, another for the game itself, and a third for reporting the player's progress. The `Scene` class is easy to extend, and your game can have as many as you need. Simply write your main loop code to control which scenes are available when.

The `betterAsteroids.py` game illustrates all these features. It's still not quite a complete game (sound effects and a more effective control system are obvious needs), but it does illustrate how to create a multi-state game. Take careful note of the special `carrier` object used to pass information between scenes. Most of the data is encapsulated inside individual scenes, but two pieces of information should be shared among scenes: the player's score and his/her intention to keep going. I placed both variables as attributes in a small object that preserves its value while the various scenes are created and destroyed.

```
  """ betterAsteroids.py
      Show more features of gameEngine
      demonstrates multiple states, multiple
      sprite groups, carrier object
  """

  import pygame, gameEngine, random

  class Ship(gameEngine.SuperSprite):
      """ player avatar.
          standard space controls
          arrows to turn, accel, space
          to fire
      """
      def __init__(self, scene):
          gameEngine.SuperSprite.__init__(self, scene)
          self.setImage("ship.gif")
          self.setSpeed(0)
          self.setAngle(0)

      def checkEvents(self):
          keys = pygame.key.get_pressed()
          if keys[pygame.K_LEFT]:
              self.rotateBy(5)
          if keys[pygame.K_RIGHT]:
              self.rotateBy(-5)
          if keys[pygame.K_UP]:
              self.addForce(.2, self.rotation)
          if keys[pygame.K_SPACE]:
              self.scene.bullet.fire()

  class Bullet(gameEngine.SuperSprite):
      """ bullet fired from spacecraft
      """

      def __init__(self, scene):
          gameEngine.SuperSprite.__init__(self, scene)
          self.setImage("bullet.gif")
          self.imageMaster = pygame.transform.scale       ↩
(self.imageMaster, (5, 5))
          self.setBoundAction(self.HIDE)
          self.reset()

      def fire(self):
          self.setPosition((self.scene.ship.x,            ↩
self.scene.ship.y))
          self.setSpeed(12)
          self.setAngle(self.scene.ship.rotation)

      def reset(self):
          self.setPosition ((-100, -100))
          self.setSpeed(0)

  class Rock(gameEngine.SuperSprite):
      """ asteroid. Rotates, wraps, and generally
          gets in the way
      """
      def __init__(self, scene):
          gameEngine.SuperSprite.__init__(self, scene)
          self.setImage("rock.gif")
          self.reset()

      def checkEvents(self):
          self.rotateBy(self.rotSpeed)

      def reset(self):
          """ change attributes randomly """

          #set random position
          x = random.randint(0, self.screen.get_width())
          y = random.randint(0, self.screen.get_height())
          self.setPosition((x, y))

          #set random size
          scale = random.randint(10, 40)
          self.setImage("rock.gif")
          self.imageMaster = \
              pygame.transform.scale(self.imageMaster,     ↩
(scale, scale))

          self.setSpeed(random.randint(0, 6))
          self.setAngle(random.randint(0, 360))
          self.rotSpeed = random.randint(-5, 5)

  class Game(gameEngine.Scene):
      """ primary gameplay state
          manages player, bullet, rocks
          returns score in carrier object
      """

      def __init__(self, carrier):
          gameEngine.Scene.__init__(self)
          self.carrier = carrier

          self.ship = Ship(self)
          self.bullet = Bullet(self)
          self.numRocks = 10
          self.points = 2000

          self.lblPoints = gameEngine.Label()
          self.lblPoints.center = (100, 30)
          self.lblPoints.text = "%d" % self.points
          self.lblPoints.fgColor = (0xFF, 0xFF, 0xFF)
          self.lblPoints.bgColor = (0, 0, 0)
          self.rocks = []
          for i in range(self.numRocks):
              self.rocks.append(Rock(self))

          self.rockGroup = self.makeSpriteGroup(self.rocks)
          self.addGroup(self.rockGroup)
          self.sprites = [self.lblPoints, self.bullet,      ↩
self.ship ]
          self.setCaption("asteroids")

      def update(self):
          rockHitShip = self.ship.collidesGroup(self.rocks)
          if rockHitShip:
              rockHitShip.reset()

          rockHitBullet = self.bullet.collidesGroup         ↩
(self.rocks)
          if rockHitBullet:
              rockHitBullet.kill()
              if len(self.rockGroup) <= 0:
                  self.stop()
                  self.carrier.score = self.points
              self.bullet.reset()

          self.points -= 1
          self.lblPoints.text = "%d" % self.points
          if self.points <= 0:
              self.stop()
              score = 0

  class Intro(gameEngine.Scene):
      """ introduction. Simply sets the
          stage
      """
      def __init__(self):
          gameEngine.Scene.__init__(self)
          instructions = gameEngine.MultiLabel()
          instructions.textLines = [
              "You are caught in an asteroid",
              "field. Use your blasters to ",
              "destroy the asteroids.",
              "",
              "Blow them up quickly for more "
              "points."]

          instructions.size = (400, 300)
          instructions.fgColor = (0xFF, 0xFF, 0xFF)
          instructions.bgColor = (0, 0, 0)
          instructions.center = (320, 200)

          self.button = gameEngine.Button()
          self.button.center = (320, 400)
          self.button.text = "Play"

          self.sprites = [instructions, self.button]
          self.setCaption("Asteroids")

      def update(self):
          if self.button.clicked:
              self.stop()

  class Report(gameEngine.Scene):
      """ reports the player's score,
          determines if player wants
          to try again
          score and data passed in
          carrier object
      """

      def __init__(self, carrier):
          gameEngine.Scene.__init__(self)

          self.carrier = carrier

          lblPoints = gameEngine.Label()
          lblPoints.center = (320, 240)
          lblPoints.fgColor = (0xFF, 0xFF, 0xFF)
          lblPoints.bgColor = (0, 0, 0)
          lblPoints.size = (300, 50)
          lblPoints.text = "Score: %d points" % carrier.score

          self.btnAgain = gameEngine.Button()
          self.btnAgain.text = "play again"
          self.btnAgain.center = (100, 400)

          self.btnQuit = gameEngine.Button()
          self.btnQuit.text = "quit"
          self.btnQuit.center = (540, 400)

          self.sprites = [lblPoints, self.btnAgain,        ↩
self.btnQuit]

      def update(self):
          if self.btnAgain.clicked:
              self.carrier.goAgain = True
```

```
                self.stop()
            if self.btnQuit.clicked:
                self.carrier.goAgain = False
                self.stop()

class Carrier(object):
    """ an object meant to hold multi-state
        data:
        score: current number of points
        goAgain: boolean continue
    """

    def __init__(self, score, goAgain):
        self.score = score
        self.goAgain = goAgain

def main():
    intro = Intro()
    intro.start()

    carrier = Carrier(20000, True)

    while carrier.goAgain:
        game = Game(carrier)
        game.start()

        report = Report(carrier)
        report.start()

if __name__ == "__main__":
    main()
```