Chapter 13

Sound, Animation, And Program Development: The Astrocrash Game

Introducing the Read Key Program

read_key.py

```
# Read Key
# Demonstrates reading the keyboard

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Ship(games.Sprite):           # A moving ship
    def update(self):
        """ Move ship based on keys pressed. """
        if games.keyboard.is_pressed(games.K_w):
            self.y -= 1
        if games.keyboard.is_pressed(games.K_s):
            self.y += 1
        if games.keyboard.is_pressed(games.K_a):
            self.x -= 1
        if games.keyboard.is_pressed(games.K_d):
            self.x += 1
```

```
def main():
    nebula_image = games.load_image("nebula.jpg",
                                    transparent = False)
    games.screen.background = nebula_image

    ship_image = games.load_image("ship.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

main()
```

The batch file: read_key.bat

```
read_key.py
pause
```

Testing for Keystrokes

- We write a class for the ship. In update(), we check for keystrokes and change the position of the ship accordingly:

```
class Ship(games.Sprite):              # A moving ship
    def update(self):
        """ Move ship based on keys pressed. """
        if games.keyboard.is_pressed(games.K_w):
            self.y -= 1
        if games.keyboard.is_pressed(games.K_s):
            self.y += 1
        if games.keyboard.is_pressed(games.K_a):
            self.x -= 1
        if games.keyboard.is_pressed(games.K_d):
            self.x += 1
```

- We use the **keyboard** object from the games module. We invoke the object's **is_pressed()**, which returns True if the key being tested for is pressed, and False if not.

- Use **is_pressed()** in if statements to test if any of the 4 keys—W, S, A, or D—is being pressed.

- If the W/S key is pressed, we decrease/increase the object's y by 1, moving the sprite up/down the screen by one pixel. If the A/D key is pressed, we decrease/increase the object's x by 1, moving the sprite left/right.

- Since multiple calls to **is_pressed()** can read simultaneous keypresses, the user can hold down multiple keys for a combined effect.

- We use the games.**K_w** constant for the W key; games.**K_s** for the S key; games.**K_a** for the A key; and games.**K_d** for the D key.

- a quick way to figure out the name of most key constants:
  - All keyboard constants begin with games.**K_**.
  - For alphabetic keys, add the key letter, in lowercase, to the end of the constant name.
  - For numeric keys, add the key number to the end of the constant name, eg, games.**K_1** for the 1 key.
  - For other keys, you can add their name in all capital letters to the end of the constant name, eg, games.**K_SPACE** for the spacebar.
- See the livewires/superwires documentation in Appendix B.

Wrapping Up the Program

Load the nebula background image, create a ship in the middle of the screen, and invoke mainloop():

```
def main():
    nebula_image = games.load_image("nebula.jpg",
                                    transparent = False)
    games.screen.background = nebula_image

    ship_image = games.load_image("ship.bmp")
    the_ship = Ship(image = ship_image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

main()
```

Introducing the Rotate Sprite Program

rotate_sprite.py

```
# Rotate Sprite
# Demonstrates rotating a sprite

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Ship(games.Sprite):
    """ A rotating ship. """
    def update(self):
        """ Rotate based on keys pressed. """
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += 1
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= 1
```

The batch file: rotate_sprite.bat

```
rotate_sprite.py
pause
```

```
        if games.keyboard.is_pressed(games.K_1):
            self.angle = 0
        if games.keyboard.is_pressed(games.K_2):
            self.angle = 90
        if games.keyboard.is_pressed(games.K_3):
            self.angle = 180
        if games.keyboard.is_pressed(games.K_4):
            self.angle = 270

def main():
    nebula_image = games.load_image("nebula.jpg",
                                    transparent = False)
    games.screen.background = nebula_image

    ship_image = games.load_image("ship.bmp")
    the_ship = Ship(image = ship_image,
      x=games.screen.width/2, y=games.screen.height/2)
    games.screen.add(the_ship)
    games.screen.mainloop()

main()
```

Using a Sprite's angle Property

- The **angle** property represents a sprite's facing in degrees.

- In update(), we check if the right/left arrow key is pressed. If yes, we add/subtract 1 to/from the object's **angle**, which rotates the sprite by 1 degree clockwise/counterclockwise.

- The next set of lines rotates the ship directly to a specific angle by assigning a new value to **angle**.

- When the user presses the 1/2/3/4 key, the code assigns 0/90/180/270 to **angle** and the sprite jumps to a rotation of 0/90/180/270 degrees (its starting orientation).

Introducing the Explosion Program

explosion.py

The batch file: explosion.bat

```
explosion.py
pause
```

```
# Explosion
# Demonstrates creating an animation

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)

nebula_image = games.load_image("nebula.jpg",
                                          transparent = 0)
games.screen.background = nebula_image

explosion_files =["explosion1.bmp", "explosion2.bmp",
                  "explosion3.bmp", "explosion4.bmp",
                  "explosion5.bmp", "explosion6.bmp",
                  "explosion7.bmp", "explosion8.bmp",
                  "explosion9.bmp"]
```

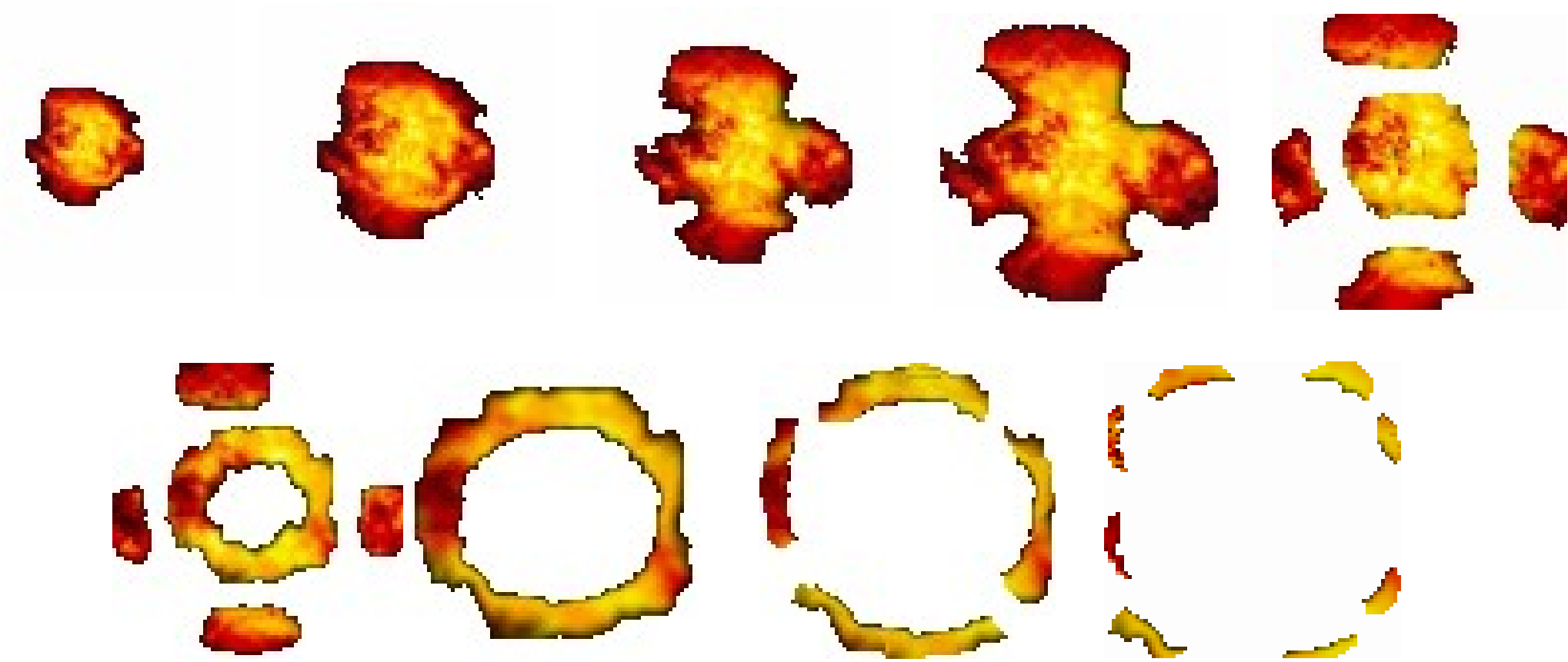
```
explosion =games.Animation(images =explosion_files,
                           x = games.screen.width/2,
                           y = games.screen.height/2,
                           n_repeats = 0, repeat_interval = 5)

games.screen.add(explosion)

games.screen.mainloop()
```

Examining the Explosion Images

- An animation is a sequence of images (also called *frames*) displayed in succession.



Creating a List of Image Files

- The constructor of the **Animation** class takes a list of image file names or a list of image objects for the sequence of images to display.

- So we create a list of image file names, which corresponds to the images:

```
explosion_files =["explosion1.bmp", "explosion2.bmp",
                  "explosion3.bmp", "explosion4.bmp",
                  "explosion5.bmp", "explosion6.bmp",
                  "explosion7.bmp", "explosion8.bmp",
                  "explosion9.bmp"]
```

Creating an Animation Object

- Create an **Animation** object and add it to the screen:

```
explosion =games.Animation(images =explosion_files,
                           x = games.screen.width/2,
                           y = games.screen.height/2,
                           n_repeats = 0, repeat_interval = 5)

games.screen.add(explosion)

games.screen.mainloop()
```

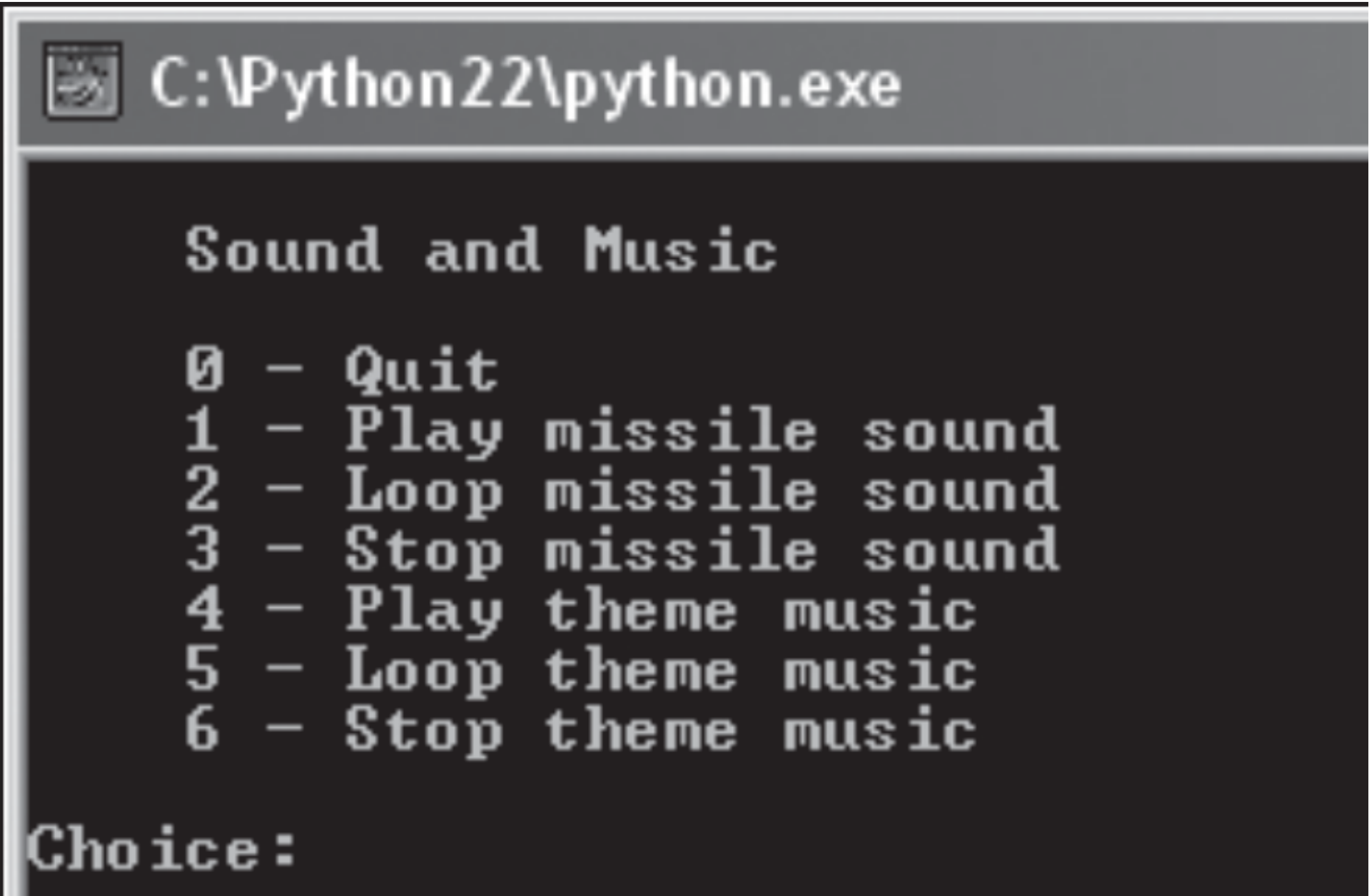
- The **Animation** class is derived from Sprite, so it inherits all of Sprite's attributes, properties, and methods.

- An animation is different from a sprite in that it has a list of images that it cycles through. You must supply a list of image file names as strings or a list of image objects for the images to be displayed.

- An object’s **n_repeats** attribute represents how many times the animation is displayed. 0 means that the animation will loop forever. The default value of **n_repeats** is 0.

- An object’s **repeat_interval** attribute represents the delay between successive images. A higher/lower number means a longer/shorter delay between frames, resulting in a slower/faster animation.

Introducing the Sound & Music Program



sound_and_music.py

```
# Sound and Music
# Demonstrates playing sound and music files

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)

# load a sound file
missile_sound = games.load_sound("missile.wav")

# load the music file
games.music.load("theme.mid")

choice = None
while choice != "0":
```

The batch file: sound_and_music.bat

```
sound_and_music.py
pause
```

```
print(
"""
Sound and Music

0 - Quit
1 - Play missile sound
2 - Loop missile sound
3 - Stop missile sound
4 - Play theme music
5 - Loop theme music
6 - Stop theme music
"""
)

choice = input("Choice: ")
print()

# exit
if choice == "0":
    print("Good-bye.")
```

```
# play missile sound
elif choice == "1":
    missile_sound.play()
    print("Playing missile sound.")

# loop missile sound
elif choice == "2":
    loop = int(input("Loop how many extra times?”+\
                                “ (-1 = forever): "))
    missile_sound.play(loop)
    print("Looping missile sound.")

# stop missile sound
elif choice == "3":
    missile_sound.stop()
    print("Stopping missile sound.")

# play theme music
elif choice == "4":
    games.music.play()
    print("Playing theme music.")
```

```
    # loop theme music
    elif choice == "5":
        loop = int(input("Loop how many extra times?”+\
                                 “ (-1 = forever): "))
        games.music.play(loop)
        print("Looping theme music.")

    # stop theme music
    elif choice == "6":
        games.music.stop()
        print("Stopping theme music.")

    # some unknown choice
    else:
        print("\nSorry, but",choice,"isn't a valid choice.")

input("\n\nPress the enter key to exit.")
```

Loading a Sound

- Create a sound object for use in a program by loading a **WAV** file, by using the games function **load_sound()**:

```
missile_sound = games.load_sound("missile.wav")
```

- You can only load WAV files with **load_sound()**.

- Load the music file:

```
games.music.load("theme.mid")
```

- Talk about this part later.

Playing a Sound

- Write a menu system:

```
choice = None
while choice != "0":
    print(
    """

    ...

    """
    )
    choice = input("Choice: ")
    print()

    if choice == "0":
        print("Good-bye.")
```

- If the user enters 0, the program says good-bye and exits.

```
elif choice == "1":
    missile_sound.play()
    print("Playing missile sound.")
```

- To play the sound once, we invoke the sound object's **play()** method. When a sound plays, it takes up 1 of the 8 available sound channels.

- To play a sound, you need at least 1 open sound channel. Once all 8 sound channels are in use, invoking a sound object's **play()** method has no effect.

Looping a Sound

You can loop a sound by passing the number of additional times you want the sound played to the object's play():

```
elif choice == "2":
    loop = int(input("Loop how many extra times?”+\
                                “ (-1 = forever): "))
    missile_sound.play(loop)
    print("Looping missile sound.")
```

Stopping a Sound

You stop a sound object from playing by invoking its **stop()** method. This stops the particular sound on all channels that it's playing:

```
    elif choice == "3":
        missile_sound.stop()
        print("Stopping missile sound.")
```

Working with Music

- In livewires/superwires, music is handled differently than sound.

- There is only **one** music channel, so only 1 file can be designated as the current music file at any given time.

- The music channel accepts many different types of sound files, including WAV, MP3, OGG, and MIDI.

- The code accesses **music** from games:

```
games.music.load("theme.mid")
```

- You load a music file by calling games.**music.load()** and passing it the music file name as a string.

- You have only 1 available music track. So, if you load a new music file, it replaces the current one.

Playing Music

```
elif choice == "4":
    games.music.play()
    print("Playing theme music.")
```

As a result, the computer plays the music, theme.mid. If you don’t pass any values to games.**music.play()**, the music plays once.

Looping Music

Loop the music by passing the number of additional times you want the music played to play():

```
    elif choice == "5":
        loop = int(input("Loop how many extra times?”+\
                         “ (-1 = forever): "))
        games.music.play(loop)
        print("Looping theme music.")
```

Stopping Music

```
    elif choice == "6":
        games.music.stop()
        print("Stopping theme music.")
```

Stop the current music from playing by calling games.**music.stop()**.

Planning the Astrocrash Game

A list of features:

- The ship rotates/thrusts forward based on keystrokes.
- The ship fire missiles based on a keystroke.
- Asteroids floats at different velocities. Smaller asteroids generally have higher velocities than larger ones.
- The ship, any missiles, and any asteroids should “wrap around” the screen—if they move beyond a screen boundary, they should appear at the opposite boundary.
- If a missile hits another object, it destroys the other object and itself in an explosion.
- If the ship hits any other object on the screen, it destroy the other object and itself in an explosion.

- If the ship is destroyed, the game is over.
- If a large asteroid is destroyed, 2 new middle ones are produced. If a middle asteroid is destroyed, 2 new small ones are produced. If A small asteroid is destroyed, no new asteroids are produced.
- Every time a player destroys an asteroid, his/her score increases. Smaller asteroids are worth more points than larger ones.
- The player’s score is displayed in the upper-right corner of the screen.
- Once all of the asteroids have been destroyed, a new, larger wave of asteroids should be created.

Game Classes

- Ship
- Missile
- Asteroid
- Explosion

Ship, Missile, and Asteroid will be derived from games.Sprite while Explosion will be derived from games.Animation.

Game Assets

Since the game includes sound, music, sprites, animation, we need to create some multimedia files:

- An image file for the ship
- An image file for the missiles
- Three image files, one for each size of asteroid
- A series of image files for an explosion
- A sound file for the thrusting of the ship
- A sound file for the firing of a missile
- A sound file for the explosion of an object
- A music file for the theme

Introducing the Astrocrash01 Program

astrocrash01.py

The batch file: astrocrash01.bat

```
astrocrash01.py
pause
```

```
# Astrocrash01
# Get asteroids moving on the screen

import random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
                MEDIUM : games.load_image("med.bmp"),
                 LARGE  : games.load_image(“big.bmp") }

    SPEED = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width

def main():                 # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    for i in range(8):   # create 8 asteroids
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
               Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    games.screen.mainloop()
# kick it off!
main()
```

The Asteroid Class

- The Asteroid class is used for creating moving asteroids:

```
class Asteroid(games.Sprite):
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    Images = {SMALL : games.load_image("small.bmp"),
              MEDIUM : games.load_image("med.bmp"),
                LARGE  : games.load_image(“big.bmp") }
    SPEED = 2
```

- Define class constants for the 3 different asteroid sizes: SMALL, MEDIUM, and LARGE.

- Then create a dictionary with the sizes and corresponding asteroid image objects. This way, we can use a size constant to look up the corresponding image object.

The __init()__ Method

```
def __init__(self, x, y, size):
    super(Asteroid, self).__init__(
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size
```

- size represents the size of the asteroid: Asteroid.SMALL, Asteroid.MEDIUM, or Asteroid.LARGE.

- Based on size, the correct image for the new asteroid is passed along to Sprite's constructor. Same as x and y.

- The velocity components are random, but smaller asteroids have the potential to move faster than larger ones.

The update() Method

- update() keeps an asteroid in play by wrapping it around the screen:

```
def update(self):

    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height

    if self.left > games.screen.width:
        self.right = 0

    if self.right < 0:
        self.left = games.screen.width
```

The main() Function

- main() sets the nebula background and creates 8 asteroids at random screen locations:

```
def main():          # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    for i in range(8):   # create 8 asteroids
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL, \
                              Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    games.screen.mainloop()
# kick it off!
main()
```

Introducing the Astrocrash02 Program

astrocrash02.py

The batch file: astrocrash02.bat

```
astrocrash02.py
pause
```

```
# Astrocrash02
# Get asteroids moving on the screen

import random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
                MEDIUM : games.load_image("med.bmp"),
                  LARGE  : games.load_image("big.bmp") }

    SPEED = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width
```

```
class Ship(games.Sprite):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    ROTATION_STEP = 3

    def update(self):    # Rotate based on keys pressed.
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

```
def main():          # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image
```

```
    # create 8 asteroids
    for i in range(8):
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
               Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(image = Ship.image,
                    x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

# kick it off!
main()
```

The Ship Class

```
class Ship(games.Sprite):
    image = games.load_image("ship.bmp")
    ROTATION_STEP = 3

    def update(self):
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

This class is similar to the Rotate Sprite program. The differences are

(1) we load the image of the ship and assign the resulting image object to the class variable image;

(2) we use the class constant ROTATION_STEP for the number of degrees by which the ship rotates.

Instantiating a Ship Object

Instantiate a Ship object and add it to the screen in main():

```
the_ship = Ship(image = Ship.image,
                x = games.screen.width/2,
                y = games.screen.height/2)
games.screen.add(the_ship)
```

Introducing the Astrocrash03 Program

astrocrash03.py

```
# Astrocrash03
# Get ship moving
```

The batch file: astrocrash03.bat

```
astrocrash03.py
pause
```

```
import math, random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
            MEDIUM : games.load_image("med.bmp"),
              LARGE  : games.load_image("big.bmp") }

    SPEED = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width

class Ship(games.Sprite):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03

    def update(self):
        """ Rotate and thrust based on keys pressed. """
        # rotate based on left and right arrow keys
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

```
# apply thrust based on up arrow key
if games.keyboard.is_pressed(games.K_UP):
    Ship.sound.play()

    # change velocity components by ship's angle
    angle = self.angle * math.pi / 180    # to radians
    self.dx += Ship.VELOCITY_STEP * \
                    math.sin(angle)
    self.dy += Ship.VELOCITY_STEP * \
                   -math.cos(angle)

# wrap the ship around screen
if self.top > games.screen.height:
                            self.bottom = 0
if self.bottom < 0:
                            self.top = games.screen.height
if self.left > games.screen.width:
                            self.right = 0
if self.right < 0:
                            self.left = games.screen.width
```

```
def main():
    # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    for i in range(8):                  # create 8 asteroids
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
                              Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(image = Ship.image,
     x=games.screen.width/2, y=games.screen.height/2)
    games.screen.add(the_ship)
    games.screen.mainloop()

# kick it off!
main()
```

Importing the math Module

import math, random

The **math** module contains a bunch of mathematical functions and constants.

Adding Ship Class Variable and Constant

- Create a class constant, VELOCITY_STEP, for altering the ship's velocity:

```
VELOCITY_STEP = .03
```

- A higher number would make the ship accelerate faster, while a lower number would make the ship accelerate more slowly.

- Add a new class variable, sound, for the thrusting sound of the ship:

```
sound = games.load_sound("thrust.wav")
```

Modifying Ship's update() Method

- Add code to Ship's update() to move the ship. Check to see if the player is pressing the up arrow key. If so, play the thrusting sound:

```
if games.keyboard.is_pressed(games.K_UP):
    Ship.sound.play()
```

- Then alter the ship's velocity components (the Ship object's dx and dy). Get the angle of the ship, converted to radians:

```
angle = self.angle * math.pi / 180
```

- Figure out how much to change each velocity component using the math module's **sin()** and **cos()** functions:

```
self.dx += Ship.VELOCITY_STEP *  math.sin(angle)
self.dy += Ship.VELOCITY_STEP * -math.cos(angle)
```

- Then handle the screen boundaries as we did with the asteroids:

```
# wrap the ship around screen
if self.top > games.screen.height:
    self.bottom = 0

if self.bottom < 0:
    self.top = games.screen.height

if self.left > games.screen.width:
    self.right = 0

if self.right < 0:
    self.left = games.screen.width
```

- Repeated chunks of code bloat programs and make them harder to maintain. When you see repeated code, it's often time for a new function or class.

Introducing the Astrocrash04 Program

astrocrash04.py

The batch file: astrocrash04.bat

```
astrocrash04.py
pause
```

```
# Astrocrash04
# Get ship firing missiles

import math, random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
            MEDIUM : games.load_image("med.bmp"),
              LARGE  : games.load_image("big.bmp") }

    SPEED = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width

class Ship(games.Sprite):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03

    def update(self):
        """ Rotate and thrust based on keys pressed. """
        # rotate based on left and right arrow keys
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

```
        # apply thrust based on up arrow key
        if games.keyboard.is_pressed(games.K_UP):
            Ship.sound.play()

            # change velocity components by ship's angle
            angle = self.angle * math.pi / 180    # to radians
            self.dx += Ship.VELOCITY_STEP * \
                           math.sin(angle)
            self.dy += Ship.VELOCITY_STEP * \
                          -math.cos(angle)

        # wrap the ship around screen
        if self.top > games.screen.height:
                                    self.bottom = 0
        if self.bottom < 0:
                                    self.top = games.screen.height
        if self.left > games.screen.width:
                                    self.right = 0
        if self.right < 0:
                                    self.left = games.screen.width
```

```
        # fire missile if spacebar pressed
        if games.keyboard.is_pressed(games.K_SPACE):
            new_missile = Missile(self.x, self.y, self.angle)
            games.screen.add(new_missile)

class Missile(games.Sprite):
    """ A missile launched by the player's ship. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40

    def __init__(self, ship_x, ship_y, ship_angle):
        """ Initialize missile sprite. """
        Missile.sound.play()

        # convert to radians
        angle = ship_angle * math.pi / 180
```

```
        # calculate missile's starting position
        buffer_x = Missile.BUFFER * math.sin(angle)
        buffer_y = Missile.BUFFER * -math.cos(angle)
        x = ship_x + buffer_x
        y = ship_y + buffer_y

        # calculate missile's velocity components
        dx = Missile.VELOCITY_FACTOR * \
                                          math.sin(angle)
        dy = Missile.VELOCITY_FACTOR * \
                                         -math.cos(angle)
        # create the missile
        super(Missile, self).__init__(image = \
              Missile.image, x = x, y = y, dx = dx, dy = dy)
        self.lifetime = Missile.LIFETIME

    def update(self):       #Move the missile
        # if lifetime is up, destroy the missile
        self.lifetime -= 1
        if self.lifetime == 0:
            self.destroy()
```

```
        # wrap the missile around screen
        if self.top > games.screen.height:
            self.bottom = 0

        if self.bottom < 0:
            self.top = games.screen.height

        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width
```

```
def main():
    # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image
```

```
    # create 8 asteroids
    for i in range(8):
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
                              Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(image = Ship.image,
                                    x = games.screen.width/2,
                                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

# kick it off!
main()
```

Modifying Ship's update() Method

- Modify Ship's update() so that a ship can fire missiles. If the player presses the spacebar, a new missile is created:

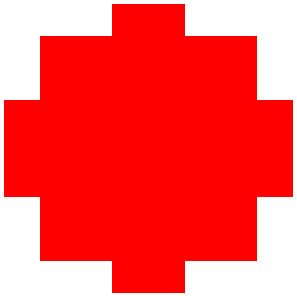
```
if games.keyboard.is_pressed(games.K_SPACE):
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
```

- in order to instantiate a new object from Missile(self.x, self.y, self.angle), we need to write a Missile class.

The Missile Class

- The Missile class is for the missiles that the ship fires:

```
class Missile(games.Sprite):
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40
```



- BUFFER represents the distance from the ship that a new missile is created. VELOCITY_FACTOR affects how fast the missile travels. LIFETIME represents how long the missile exists before it disappears.

The __init__() Method

```
def __init__(self, ship_x, ship_y, ship_angle):
```

- The method needs the values to determine 2 things: exactly where the missile first appears and its velocity components.

- Play the missile-firing sound effect:

```
Missile.sound.play()
```

- Calculate to know the new missile's starting location:

```
angle = ship_angle * math.pi / 180

buffer_x = Missile.BUFFER *  math.sin(angle)
buffer_y = Missile.BUFFER * -math.cos(angle)
x = ship_x + buffer_x
y = ship_y + buffer_y
```

- Calculate the missile's velocity components:

```
dx = Missile.VELOCITY_FACTOR * \
                                    math.sin(angle)
dy = Missile.VELOCITY_FACTOR * \
                                   -math.cos(angle)
```

- Invoke the Sprite constructor for the object:

```
super(Missile, self).__init__(image = \
      Missile.image, x = x, y = y, dx = dx, dy = dy)
```

- Give the Missile object a lifetime attribute so that the object won't be around forever:

```
self.lifetime = Missile.LIFETIME
```

The update() Method

```
def update(self):
    self.lifetime -= 1
    if self.lifetime == 0:
        self.destroy()
```

- Counts down the life of the missile. lifetime is decremented. When it reaches 0, the Missile object destroys itself.

- Wrap the missile around the screen:

```
    if self.top > games.screen.height:
                                self.bottom = 0
    if self.bottom < 0:
                                self.top = games.screen.height
    if self.left > games.screen.width:
                                self.right = 0
    if self.right < 0:
                                self.left = games.screen.width
```

Introducing the Astrocrash05 Program

astrocrash05.py

The batch file: astrocrash05.bat

```
astrocrash05.py
pause
```

```
# Astrocrash05
# Limiting missile fire rate

import math, random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
            MEDIUM : games.load_image("med.bmp"),
              LARGE  : games.load_image("big.bmp") }

    SPEED = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width


class Ship(games.Sprite):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03
    MISSILE_DELAY = 25

    def __init__(self, x, y):
        """ Initialize ship sprite. """
        super(Ship, self).__init__(image = Ship.image,
                                   x = x, y = y)
        self.missile_wait = 0
```

```
def update(self):
    """ Rotate and thrust based on keys pressed. """
    # rotate based on left and right arrow keys
    if games.keyboard.is_pressed(games.K_LEFT):
        self.angle -= Ship.ROTATION_STEP
    if games.keyboard.is_pressed(games.K_RIGHT):
        self.angle += Ship.ROTATION_STEP

    # apply thrust based on up arrow key
    if games.keyboard.is_pressed(games.K_UP):
        Ship.sound.play()

        # change velocity components by ship's angle
        angle = self.angle * math.pi / 180    # to radians
        self.dx += Ship.VELOCITY_STEP * \
                                    math.sin(angle)
        self.dy += Ship.VELOCITY_STEP * \
                                   -math.cos(angle)
```

```
# wrap the ship around screen
if self.top > games.screen.height:
    self.bottom = 0

if self.bottom < 0:
    self.top = games.screen.height

if self.left > games.screen.width:
    self.right = 0

if self.right < 0:
    self.left = games.screen.width
```

```
# decrease waiting until the ship can fire next
if self.missile_wait > 0:
    self.missile_wait -= 1
```

```
        # fire missile if space pressed and wait is over
        if games.keyboard.is_pressed(games.K_SPACE) \
                                        and self.missile_wait == 0:
            new_missile = Missile(self.x, self.y, self.angle)
            games.screen.add(new_missile)
            self.missile_wait = Ship.MISSILE_DELAY

class Missile(games.Sprite):
    """ A missile launched by the player's ship. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40

    def __init__(self, ship_x, ship_y, ship_angle):
        """ Initialize missile sprite. """
        Missile.sound.play()

        # convert to radians
        angle = ship_angle * math.pi / 180
```

```
# calculate missile's starting position
buffer_x = Missile.BUFFER * math.sin(angle)
buffer_y = Missile.BUFFER * -math.cos(angle)
x = ship_x + buffer_x
y = ship_y + buffer_y

# calculate missile's velocity components
dx = Missile.VELOCITY_FACTOR * \
                                math.sin(angle)
dy = Missile.VELOCITY_FACTOR * \
                               -math.cos(angle)

# create the missile
super(Missile, self).__init__(image = \
      Missile.image, x = x, y = y, dx = dx, dy = dy)
self.lifetime = Missile.LIFETIME
```

```
def update(self):
    """ Move the missile. """
    # if lifetime is up, destroy the missile
    self.lifetime -= 1
    if self.lifetime == 0:
        self.destroy()

    # wrap the missile around screen
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height

    if self.left > games.screen.width:
        self.right = 0

    if self.right < 0:
        self.left = games.screen.width
```

```
def main():                     # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    for i in range(8):          # create 8 asteroids
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
                              Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

# kick it off!
main()
```

Creating Ship's Constructor Method

- Add a class constant, MISSILE_DELAY, to Ship to force a delay between missile firings. It represents the delay a player must wait between missile firings:

```
MISSILE_DELAY = 25
```

- Create a constructor method for the class:

```
def __init__(self, x, y):
    super(Ship, self).__init__(image = Ship.image,
                               x = x, y = y)
    self.missile_wait = 0
```

- The method accepts values for the x- and y-coordinates of the new ship and passes those off to the superclass of Ship, games.Sprite.

- missile_wait is used to count down the delay until the player can fire the next missile.

Modifying Ship's update() Method

- Add some code to Ship's update() to decrement an object's missile_wait, counting it down to 0:

```
if self.missile_wait > 0:
    self.missile_wait -= 1
```

- Change the missile firing code to

```
if games.keyboard.is_pressed(games.K_SPACE) \
                                and self.missile_wait == 0:
    new_missile = Missile(self.x, self.y, self.angle)
    games.screen.add(new_missile)
    self.missile_wait = Ship.MISSILE_DELAY
```

- When the player presses the spacebar, the countdown must be complete (missile_wait must be 0) before the ship will fire a new missile. Once a missile is fired, we reset missile_wait to MISSILE_DELAY to begin the countdown again.

Introducing the Astrocrash06 Program

astrocrash06.py

The batch file: astrocrash06.bat

```
astrocrash06.py
pause
```

```
# Astrocrash06
# Handling collisions

import math, random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Asteroid(games.Sprite):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
            MEDIUM : games.load_image("med.bmp"),
              LARGE  : games.load_image("big.bmp") }

    SPEED = 2
    SPAWN = 2
```

```
def __init__(self, x, y, size):
    """ Initialize asteroid sprite. """
    super(Asteroid, self).__init__( \
        image = Asteroid.images[size], x = x, y = y,
        dx = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size,
        dy = random.choice([1, -1]) * Asteroid.SPEED \
            * random.random()/size)

    self.size = size

def update(self):
    """ Wrap around screen. """
    if self.top > games.screen.height:
        self.bottom = 0

    if self.bottom < 0:
        self.top = games.screen.height
```

```
if self.left > games.screen.width:
    self.right = 0

if self.right < 0:
    self.left = games.screen.width
```

```
def die(self):
    """ Destroy asteroid. """
    # if asteroid isn't small, replace with 2 smaller
    if self.size != Asteroid.SMALL:
        for i in range(Asteroid.SPAWN):
            new_asteroid = Asteroid(x= self.x, y= self.y,
                                    size = self.size - 1)
            games.screen.add(new_asteroid)
    self.destroy()
```

```
class Ship(games.Sprite):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03
    MISSILE_DELAY = 25

    def __init__(self, x, y):
        """ Initialize ship sprite. """
        super(Ship, self).__init__(image = Ship.image,
                                   x = x, y = y)
        self.missile_wait = 0

    def update(self):
        """ Rotate and thrust based on keys pressed. """
        # rotate based on left and right arrow keys
        if games.keyboard.is_pressed(games.K_LEFT):
            self.angle -= Ship.ROTATION_STEP
        if games.keyboard.is_pressed(games.K_RIGHT):
            self.angle += Ship.ROTATION_STEP
```

```
        # apply thrust based on up arrow key
        if games.keyboard.is_pressed(games.K_UP):
            Ship.sound.play()

            # change velocity components by ship's angle
            angle = self.angle * math.pi / 180  # to radians
            self.dx += Ship.VELOCITY_STEP * \
                                        math.sin(angle)
            self.dy += Ship.VELOCITY_STEP * \
                                       -math.cos(angle)

        # wrap the ship around screen
        if self.top > games.screen.height:
                                    self.bottom = 0
        if self.bottom < 0:
                                    self.top = games.screen.height
        if self.left > games.screen.width:
                                    self.right = 0
        if self.right < 0:
                                    self.left = games.screen.width
```

```
        # decrease wait until the ship can fire next
        if self.missile_wait > 0:
            self.missile_wait -= 1

        # fire if spacebar pressed and wait is over
        if games.keyboard.is_pressed(games.K_SPACE) \
                                        and self.missile_wait == 0:
            new_missile = Missile(self.x, self.y, self.angle)
            games.screen.add(new_missile)
            self.missile_wait = Ship.MISSILE_DELAY

        # check if ship overlaps any other object
        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()

    def die(self):
        """ Destroy ship. """
        self.destroy()
```

```
class Missile(games.Sprite):
    """ A missile launched by the player's ship. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40

    def __init__(self, ship_x, ship_y, ship_angle):
        """ Initialize missile sprite. """
        Missile.sound.play()

        # convert to radians
        angle = ship_angle * math.pi / 180

        # calculate missile's starting position
        buffer_x = Missile.BUFFER * math.sin(angle)
        buffer_y = Missile.BUFFER * -math.cos(angle)
        x = ship_x + buffer_x
        y = ship_y + buffer_y
```

```
        # calculate missile's velocity components
        dx = Missile.VELOCITY_FACTOR * \
                                        math.sin(angle)
        dy = Missile.VELOCITY_FACTOR * \
                                       -math.cos(angle)

        # create the missile
        super(Missile, self).__init__(image = \
                Missile.image, x = x, y = y, dx = dx, dy = dy)
        self.lifetime = Missile.LIFETIME

    def update(self):
        """ Move the missile. """
        # if lifetime is up, destroy the missile
        self.lifetime -= 1
        if self.lifetime == 0:
            self.destroy()
```

```
        if self.bottom < 0:
            self.top = games.screen.height

        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width
```

```
        # check if missile overlaps any other object
        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()

    def die(self):
        """ Destroy the missile. """
        self.destroy()
```

```
def main():    # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    for i in range(8):     # create 8 asteroids
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
              Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

# kick it off!
main()
```

Modifying Missile's update() Method

```
if self.overlapping_sprites:
    for sprite in self.overlapping_sprites:
        sprite.die()
    self.die()
```

- If a missile overlaps any other objects, the other objects and the missile all have their die() called.

- die() is a new method added to Asteroid, Ship, and Missile.

Adding Missile's die() Method

- Missile, like any class in this game, needs a die() method:

```
def die(self):
    self.destroy()
```

- When a Missile object's die() is invoked, the object destroys itself.

Modifying Ship's update() Method

- Add the following code to the Ship's update() method:

```
if self.overlapping_sprites:
    for sprite in self.overlapping_sprites:
        sprite.die()
    self.die()
```

- If the ship overlaps any other objects, the other objects and the ship all have their die() called.

- This exact code also appears in Missile's update() method.

- When you see duplicate code, you should think about how to consolidate it.

Adding Ship's die() Method

- This method is the same as Missile's die() method:

```
def die(self):
    self.destroy()
```

- When a Ship object's die() method is invoked, the object destroys itself.

Adding Asteroid's die() Method

- Add class constant, SPAWN, to Asteroid. It is the number of new asteroids that an asteroid spawns when destroyed:

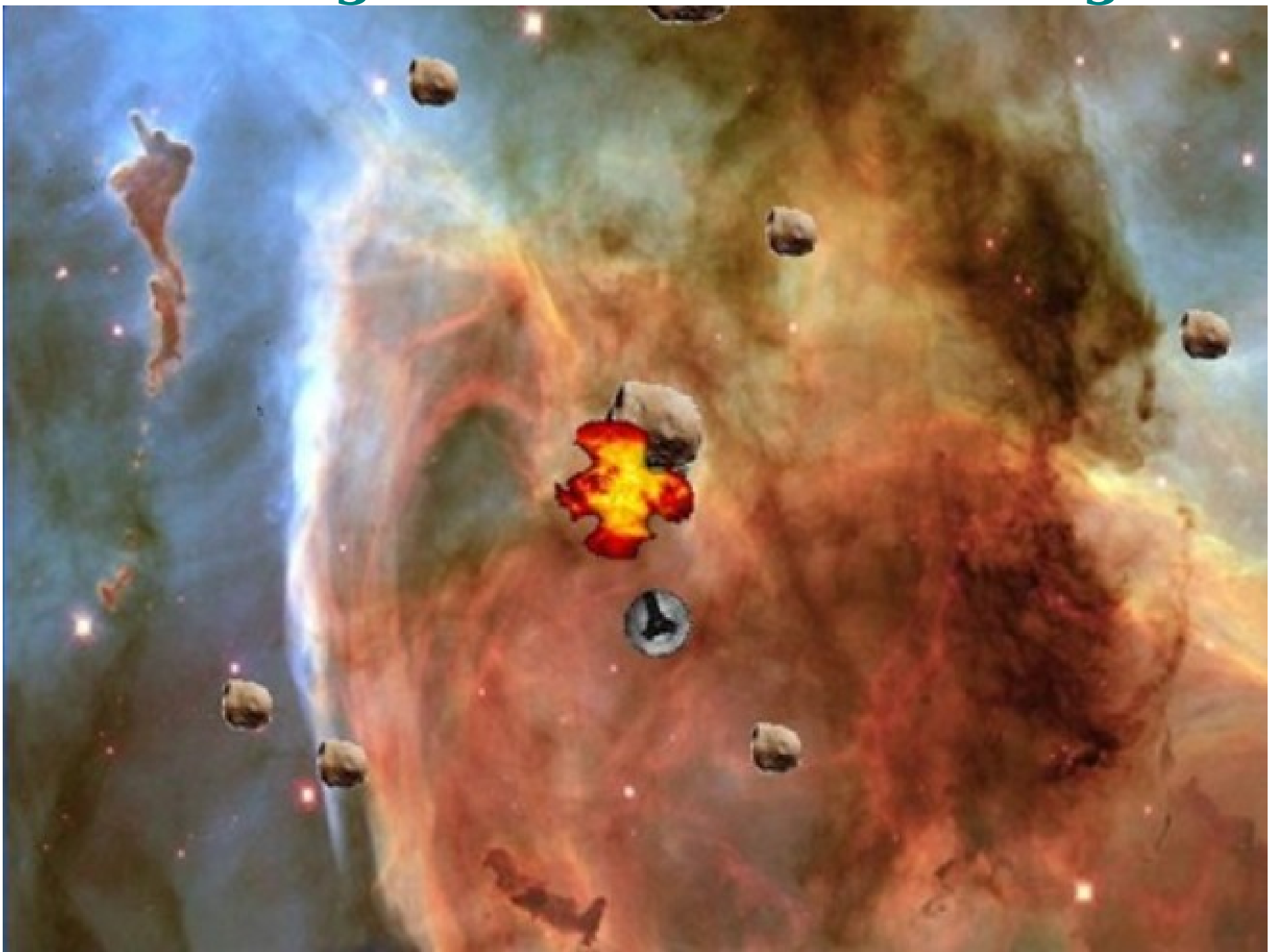
```
SPAWN = 2
```

- Asteroid's die() method is more involved than the others:

```
def die(self):
    if self.size != Asteroid.SMALL:
        for i in range(Asteroid.SPAWN):
            new_asteroid = Asteroid(x = self.x, y = self.y,
                                    size = self.size - 1)
            games.screen.add(new_asteroid)
    self.destroy()
```

- The method checks if the asteroid being destroyed isn't small. If not, 2 new smaller asteroids are created at the current location. With/without new asteroids, the current asteroid destroys itself.

Introducing the Astrocrash07 Program



astrocrash07.py

```
# Astrocrash07
# Add explosions
```

The batch file: astrocrash07.bat

```
astrocrash07.py
pause
```

```
import math, random
from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Wrapper(games.Sprite):
    """ A sprite that wraps around the screen. """
    def update(self):         #Wrap sprite around screen.
        if self.top > games.screen.height:
                                self.bottom = 0
        if self.bottom < 0:
                                self.top = games.screen.height
        if self.left > games.screen.width:
                                self.right = 0
        if self.right < 0:
                                self.left = games.screen.width
```

```
    def die(self):
        """ Destroy self. """
        self.destroy()

class Collider(Wrapper):
    """ A Wrapper that collide with another object. """
    def update(self):
        """ Check for overlapping sprites. """
        super(Collider, self).update()

        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()

    def die(self):
        """ Destroy self and leave explosion behind. """
        new_explosion = Explosion(x = self.x, y = self.y)
        games.screen.add(new_explosion)
        self.destroy()
```

```
class Asteroid(Wrapper):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
                MEDIUM : games.load_image("med.bmp"),
                  LARGE  : games.load_image("big.bmp") }

    SPEED = 2
    SPAWN = 2

    def __init__(self, x, y, size):       # Initialize asteroid
        super(Asteroid, self).__init__( \
            image = Asteroid.images[size], x = x, y = y,
            dx = random.choice([1, -1]) * Asteroid.SPEED \
                 * random.random()/size,
            dy = random.choice([1, -1]) * Asteroid.SPEED \
                 * random.random()/size)

        self.size = size
```

```
    def die(self):
        """ Destroy asteroid. """
        # if not small, replace with 2 smaller asteroids
        if self.size != Asteroid.SMALL:
            for i in range(Asteroid.SPAWN):
                new_asteroid =Asteroid(x = self.x, y = self.y,
                                       size = self.size - 1)
                games.screen.add(new_asteroid)

        super(Asteroid, self).die()

class Ship(Collider):
    """ The player's ship. """
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03
    MISSILE_DELAY = 25
```

```
def __init__(self, x, y):
    """ Initialize ship sprite. """
    super(Ship, self).__init__(image = Ship.image,
                               x = x, y = y)
    self.missile_wait = 0

def update(self):
    """ Rotate, thrust, fire based on keys pressed. """
    super(Ship, self).update()

    # rotate based on left and right arrow keys
    if games.keyboard.is_pressed(games.K_LEFT):
        self.angle -= Ship.ROTATION_STEP
    if games.keyboard.is_pressed(games.K_RIGHT):
        self.angle += Ship.ROTATION_STEP

    # apply thrust based on up arrow key
    if games.keyboard.is_pressed(games.K_UP):
        Ship.sound.play()
```

```
        # change velocity components by ship's angle
        angle = self.angle * math.pi / 180    # to radians
        self.dx += Ship.VELOCITY_STEP * \
                                    math.sin(angle)
        self.dy += Ship.VELOCITY_STEP * \
                                   -math.cos(angle)

    # decrease wait until the ship can fire next
    if self.missile_wait > 0:
        self.missile_wait -= 1

    # fire if spacebar pressed and wait is over
    if games.keyboard.is_pressed(games.K_SPACE) \
                                    and self.missile_wait == 0:
        new_missile = Missile(self.x, self.y, self.angle)
        games.screen.add(new_missile)
        self.missile_wait = Ship.MISSILE_DELAY
```

```
class Missile(Collider):
    """ A missile launched by the player's ship. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40

    def __init__(self, ship_x, ship_y, ship_angle):
        """ Initialize missile sprite. """
        Missile.sound.play()

        # convert to radians
        angle = ship_angle * math.pi / 180

        # calculate missile's starting position
        buffer_x = Missile.BUFFER * math.sin(angle)
        buffer_y = Missile.BUFFER * -math.cos(angle)
        x = ship_x + buffer_x
        y = ship_y + buffer_y
```

```
        # calculate missile's velocity components
        dx = Missile.VELOCITY_FACTOR * \
                                        math.sin(angle)
        dy = Missile.VELOCITY_FACTOR * \
                                       -math.cos(angle)

        # create the missile
        super(Missile, self).__init__(image = \
              Missile.image, x = x, y = y, dx = dx, dy = dy)
        self.lifetime = Missile.LIFETIME

    def update(self):
        """ Move the missile. """
        super(Missile, self).update()

        # if lifetime is up, destroy the missile
        self.lifetime -= 1
        if self.lifetime == 0:
            self.destroy()
```

```
class Explosion(games.Animation):
    """ Explosion animation. """
    sound = games.load_sound("explosion.wav")
    images = ["explosion1.bmp", "explosion2.bmp",
              "explosion3.bmp", "explosion4.bmp",
              "explosion5.bmp", "explosion6.bmp",
              "explosion7.bmp", "explosion8.bmp",
              "explosion9.bmp"]

    def __init__(self, x, y):
        super(Explosion, self).__init__(images = \
                Explosion.images, x = x, y = y,
                repeat_interval = 4, n_repeats = 1,
                                    is_collideable = False)
        Explosion.sound.play()
```

```
def main():
    # establish background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image
```

```
    # create 8 asteroids
    for i in range(8):
        x = random.randrange(games.screen.width)
        y = random.randrange(games.screen.height)
        size = random.choice([Asteroid.SMALL,
            Asteroid.MEDIUM, Asteroid.LARGE])
        new_asteroid = Asteroid(x = x, y = y, size = size)
        games.screen.add(new_asteroid)

    # create the ship
    the_ship = Ship(x = games.screen.width/2,
                    y = games.screen.height/2)
    games.screen.add(the_ship)

    games.screen.mainloop()

# kick it off!
main()
```

The Wrapper Class

- Start with the behind-the-scenes work. Create a new class, Wrapper, based on games.Sprite. Wrapper's update() automatically wraps an object around the screen:

```
class Wrapper(games.Sprite):
    def update(self):
        if self.top > games.screen.height:
            self.bottom = 0

        if self.bottom < 0:
            self.top = games.screen.height

        if self.left > games.screen.width:
            self.right = 0

        if self.right < 0:
            self.left = games.screen.width
```

- We’ve seen this code several times already. It wraps a sprite around the screen. Now, if we base the other classes in the game on Wrapper, its update() can keep instances of those other classes on the screen—and the code only has to exist in one place!

- Finish the class up with die() that destroys the object:

```
def die(self):
    self.destroy()
```

The Collider Class

- Both Ship and Missile share the same collision handling, so we create Collider (based on Wrapper) for objects that wrap around the screen and that can collide with other objects:

```
class Collider(Wrapper):
    def update(self):
        super(Collider, self).update()

        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()
```

- The 1st thing is to invoke its superclass's update(), ie, Wrapper's update(), to keep the object on the screen.

- Then check for collisions. If the object overlaps any others, call die() for the other objects and for the object.

The die() Method

- Have a die() method for the class, since all Collider objects will do the same thing when they die—create an explosion and destroy themselves:

```
def die(self):
    new_explosion = Explosion(x = self.x, y = self.y)
    games.screen.add(new_explosion)
    self.destroy()
```

- Explosion is a new class whose objects are explosion animations.

Modifying the Asteroid Class

- Modify Asteroid so that the class is based on Wrapper:

class Asteroid(Wrapper):

- Asteroid now inherits update() from Wrapper, so we cut Asteroid's own update().

- The only other thing is to change Asteroid's die(). We replace self.die() with

super(Asteroid, self).die()

- if we ever change Wrapper's die(), Asteroid will automatically reap the benefits.

Modifying the Ship Class

- Modify Ship so that the class is based on Collider:

class Ship(Collider):

- In Ship's update(), we add

super(Ship, self).update()

- Since Collider's update() handles collision, we cut the code for collision detection from Ship's update(). Since Collider's update() invokes Wrapper's update(), we cut the screen wrapping code from Ship's update(), too. We also cut Ship's die(), as the class inherits Collider's version.

Modifying the Missile Class

- Modify Missile so that the class is based on Collider:

class Missile(Collider):

- In Missile's update(), we add

super(Missile, self).update()

- Since Collider's update() handles collision, we cut the code for collision detection from Missile's update(). Collider's update() invokes Wrapper's update() method, so we cut the screen wrapping code from Missile's update(), too. We also cut Missile's die(), as the class inherits Collider's version.

The Explosion Class

- Since we want to create animated explosions, we write an Explosion class based on games.Animation:

```
class Explosion(games.Animation):
    """ Explosion animation. """
    sound = games.load_sound("explosion.wav")
    images = ["explosion1.bmp", "explosion2.bmp",
              "explosion3.bmp", "explosion4.bmp",
              "explosion5.bmp", "explosion6.bmp",
              "explosion7.bmp", "explosion8.bmp",
              "explosion9.bmp"]
```

- Define sound for the sound effect of an explosion. Define images for the list of image file names for the 9 frames of the explosion animation.

- The Explosion constructor:

```
def __init__(self, x, y):
    super(Explosion, self).__init__(images = \
            Explosion.images, x = x, y = y,
            repeat_interval = 4, n_repeats = 1,
                                is_collideable = False)
    Explosion.sound.play()
```

- x and y represent the screen coordinates for the explosion.
- Pass 1 to n_repeats so that the animation plays just once.
- pass 4 to repeat_interval so that the speed of the animation looks right.
- Pass False to is_collideable so that the explosion animation doesn’t count as a collision for other sprites that might happen to overlap it.
- Play the explosion sound effect with Explosion.sound.play() at the end.

Introducing the Astrocrash08 Program

astrocrash08.py

```
# Astrocrash08
# Add Game object for complete program

import math, random
from superwires import games, color

games.init(screen_width = 640, screen_height = 480,
                                                fps = 50)
class Wrapper(games.Sprite):
    """ A sprite that wraps around the screen. """
    def update(self):          #Wrap sprite around screen.
        if self.top > games.screen.height:
                                self.bottom = 0
        if self.bottom < 0:
                                self.top = games.screen.height
        if self.left > games.screen.width:
                                self.right = 0
        if self.right < 0:
                                self.left = games.screen.width
```

```
    def die(self):
        """ Destroy self. """
        self.destroy()

class Collider(Wrapper):
    """ A Wrapper that collide with another object. """
    def update(self):
        """ Check for overlapping sprites. """
        super(Collider, self).update()

        if self.overlapping_sprites:
            for sprite in self.overlapping_sprites:
                sprite.die()
            self.die()

    def die(self):
        """ Destroy self and leave explosion behind. """
        new_explosion = Explosion(x = self.x, y = self.y)
        games.screen.add(new_explosion)
        self.destroy()
```

```
class Asteroid(Wrapper):
    """ An asteroid which floats across the screen. """
    SMALL = 1
    MEDIUM = 2
    LARGE = 3
    images={SMALL : games.load_image("small.bmp"),
                MEDIUM : games.load_image("med.bmp"),
                  LARGE  : games.load_image("big.bmp") }

    SPEED = 2
    SPAWN = 2
    POINTS = 30

    total =  0

    def __init__(self, game, x, y, size):
        """ Initialize asteroid sprite. """
        Asteroid.total += 1
```

```
        super(Asteroid, self).__init__( \
            image = Asteroid.images[size], x = x, y = y,
            dx = random.choice([1, -1]) * Asteroid.SPEED \
                * random.random()/size,
            dy = random.choice([1, -1]) * Asteroid.SPEED \
                * random.random()/size)

        self.game = game
        self.size = size

    def die(self):
        """ Destroy asteroid. """
        Asteroid.total -= 1

        self.game.score.value += \
                                int(Asteroid.POINTS / self.size)
        self.game.score.right = games.screen.width - 10
```

```
        # if not small, replace with two smaller asteroids
        if self.size != Asteroid.SMALL:
            for i in range(Asteroid.SPAWN):
                new_asteroid = Asteroid(game = self.game,
                               x = self.x, y = self.y, size = self.size - 1)
                games.screen.add(new_asteroid)

        # if all asteroids are gone, advance to next level
        if Asteroid.total == 0:
            self.game.advance()

        super(Asteroid, self).die()

class Ship(Collider):                   # The player's ship
    image = games.load_image("ship.bmp")
    sound = games.load_sound("thrust.wav")
    ROTATION_STEP = 3
    VELOCITY_STEP = .03
    VELOCITY_MAX = 3
    MISSILE_DELAY = 25
```

The batch file: astrocrash08.bat

```
astrocrash08.py
pause
```

```
def __init__(self, game, x, y):
    """ Initialize ship sprite. """
    super(Ship, self).__init__(image = Ship.image,
                               x = x, y = y)
    self.game = game
    self.missile_wait = 0

def update(self):
    """ Rotate, thrust, fire based on keys pressed. """
    super(Ship, self).update()

    # rotate based on left and right arrow keys
    if games.keyboard.is_pressed(games.K_LEFT):
        self.angle -= Ship.ROTATION_STEP
    if games.keyboard.is_pressed(games.K_RIGHT):
        self.angle += Ship.ROTATION_STEP
```

```
# apply thrust based on up arrow key
if games.keyboard.is_pressed(games.K_UP):
    Ship.sound.play()

    # change velocity components by ship's angle
    angle = self.angle * math.pi / 180    # to radians
    self.dx += Ship.VELOCITY_STEP * \
                                  math.sin(angle)
    self.dy += Ship.VELOCITY_STEP * \
                                 -math.cos(angle)

    # cap velocity in each direction
    self.dx = min(max(self.dx,
    -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
    self.dy = min(max(self.dy,
    -Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)

# decrease wait until the ship can fire next
if self.missile_wait > 0:
    self.missile_wait -= 1
```

```
        # fire if spacebar pressed & missile wait is over
        if games.keyboard.is_pressed(games.K_SPACE) \
                                        and self.missile_wait == 0:
            new_missile = Missile(self.x, self.y, self.angle)
            games.screen.add(new_missile)
            self.missile_wait = Ship.MISSILE_DELAY

    def die(self):
        """ Destroy ship and end the game. """
        self.game.end()
        super(Ship, self).die()

class Missile(Collider):
    """ A missile launched by the player's ship. """
    image = games.load_image("missile.bmp")
    sound = games.load_sound("missile.wav")
    BUFFER = 40
    VELOCITY_FACTOR = 7
    LIFETIME = 40
```

```
def __init__(self, ship_x, ship_y, ship_angle):
    """ Initialize missile sprite. """
    Missile.sound.play()

    # convert to radians
    angle = ship_angle * math.pi / 180

    # calculate missile's starting position
    buffer_x = Missile.BUFFER * math.sin(angle)
    buffer_y = Missile.BUFFER * -math.cos(angle)
    x = ship_x + buffer_x
    y = ship_y + buffer_y

    # calculate missile's velocity components
    dx = Missile.VELOCITY_FACTOR * \
                                  math.sin(angle)
    dy = Missile.VELOCITY_FACTOR * \
                                 -math.cos(angle)
```

```
        # create the missile
        super(Missile, self).__init__(image = \
        Missile.image, x = x, y = y, dx = dx, dy = dy)
        self.lifetime = Missile.LIFETIME

    def update(self):                       # Move the missile.
        super(Missile, self).update()

        # if lifetime is up, destroy the missile
        self.lifetime -= 1
        if self.lifetime == 0:
            self.destroy()

class Explosion(games.Animation):
    """ Explosion animation. """
    sound = games.load_sound("explosion.wav")
    images = ["explosion1.bmp", "explosion2.bmp",
              "explosion3.bmp”, "explosion4.bmp",
              "explosion5.bmp", "explosion6.bmp",
              "explosion7.bmp", "explosion8.bmp",
              "explosion9.bmp"]
```

```
    def __init__(self, x, y):
        super(Explosion, self).__init__(images = \
        Explosion.images, x= x, y= y, repeat_interval = 4,
        n_repeats = 1, is_collideable = False)
        Explosion.sound.play()
```

```
class Game(object):
    """ The game itself. """
    def __init__(self):
        """ Initialize Game object. """
        self.level = 0                          # set level

        # load sound for level advance
        self.sound = games.load_sound("level.wav")

        # create score
        self.score = games.Text(value = 0, size = 30,
        color = color.white, top = 5,
        right = games.screen.width – 10,
        is_collideable = False)
        games.screen.add(self.score)
```

```
    # create player's ship
    self.ship = Ship(game = self,
                     x = games.screen.width/2,
                     y = games.screen.height/2)
    games.s creen.add(self.ship)

def play(self):
    """ Play the game. """
    # begin theme music
    games.music.load("theme.mid")
    games.music.play(-1)

    # load and set background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    # advance to level 1
    self.advance()

    # start play
    games.screen.mainloop()
```

```
def advance(self):
    """ Advance to the next game level. """
    self.level += 1

    # preserve space near ship if creating asteroids
    BUFFER = 150

    # create new asteroids
    for i in range(self.level):
        # calculate (x, y) BUFFER from the ship

        # choose min distance along x-axis and y-axis
        x_min = random.randrange(BUFFER)
        y_min = BUFFER - x_min

        # choose components  based on min distance
        x_distance = random.randrange(x_min,
        games.screen.width - x_min)
        y_distance = random.randrange(y_min,
        games.screen.height – y_min)
```

```
        # calculate location based on distance
        x = self.ship.x + x_distance
        y = self.ship.y + y_distance

        # wrap around screen, if necessary
        x %= games.screen.width
        y %= games.screen.height

        # create the asteroid
        new_asteroid = Asteroid(game = self,
        x = x, y = y, size = Asteroid.LARGE)
        games.screen.add(new_asteroid)

# display level number
level_message = games.Message(value= "Level" \
+ str(self.level), size = 40, color = color.yellow,
x = games.screen.width/2,
y = games.screen.width/10,
lifetime = 3 * games.screen.fps,
is_collideable = False)
games.screen.add(level_message)
```

```
        # play new level sound (except at first level)
        if self.level > 1:
            self.sound.play()

    def end(self):                              # End the game
        # show 'Game Over' for 5 seconds
        end_message = games.Message(value =  \
        "Game Over", size = 90, color = color.red,
        x = games.screen.width/2,
        y = games.screen.height/2,
        lifetime = 5 * games.screen.fps,
        after_death = games.screen.quit,
        is_collideable = False)
        games.screen.add(end_message)

def main():
    astrocrash = Game()
    astrocrash.play()

# kick it off!
Main()
```

Importing the color Module

- Along with games, import **color** from livewires/superwires:

from livewires import games, color

- Need the **color** module so that the "Game Over" message can be displayed in a nice, bright red color.

The Game Class

- The Game class—a new class for an object that represents the game itself.

- The game itself could certainly be an object with methods like play() to start the game, advance() to move the game to the next level, and end() to end the game.

- Designing the game as an object makes it easy for other objects to send the game messages.

- Much of the code that was in main() has been incorporated into Game.

The __init__() Method

```
class Game(object):
    def __init__(self):
        self.level = 0                          # set level

        self.sound = games.load_sound("level.wav")

        # create score
        self.score = games.Text(value = 0, size = 30,
        color = color.white, top = 5,
        right = games.screen.width – 10,
        is_collideable = False)
        games.screen.add(self.score)

        # create player's ship
        self.ship = Ship(game = self,
                         x = games.screen.width/2,
                         y = games.screen.height/2)
        games.screen.add(self.ship)
```

- level is an attribute for the current game level number. sound is an attribute for the level-advance sound effect. score is an attribute for the game score—it's a Text object that appears in the upper-right corner of the screen.

- The object's is_collideable property is False, which means that the score won't register in any collisions—so the player's ship won't "crash into" the score and explode!

- ship is an attribute for the player's ship.

The play() Method

```
def play(self):
    games.music.load("theme.mid")
    games.music.play(-1)

    # load and set background
    nebula_image = games.load_image("nebula.jpg")
    games.screen.background = nebula_image

    self.advance()

    games.screen.mainloop()
```

- The method loads the theme music and plays it so that it will loop forever. It loads the nebula image and sets it as the background. Then the method calls the Game object's own advance(), which advances the game to the next level. Then, play() invokes games.screen.mainloop() to kick off the whole game!

The advance() Method

- advance() moves the game to the next level. It increments the level number, creates a new wave of asteroids, displays the level number, and plays the level-advance sound.

- Increase the level number firstly:

```
def advance(self):
    self.level += 1
```

- Each level starts with the number of asteroids equal to the level number. So, the 1st level starts with only 1 asteroid, the 2nd with 2, and so on.

- Need to make sure that no new asteroid is created right on top of the ship. BUFFER is a constant for the amount of safe space around the ship.:

```
BUFFER = 150
```

```
# create new asteroids
for i in range(self.level):
    # calculate (x, y) BUFFER from the ship

    # choose min distance along x-axis and y-axis
    x_min = random.randrange(BUFFER)
    y_min = BUFFER - x_min

    # choose components  based on min distance
    x_distance = random.randrange(x_min,
    games.screen.width - x_min)
    y_distance = random.randrange(y_min,
    games.screen.height – y_min)

    # calculate location based on distance
    x = self.ship.x + x_distance
    y = self.ship.y + y_distance

    # wrap around screen, if necessary
    x %= games.screen.width
    y %= games.screen.height
```

```
# create the asteroid
new_asteroid = Asteroid(game = self,
x = x, y = y, size = Asteroid.LARGE)
games.screen.add(new_asteroid)
```

- Start a loop. In each iteration, create a new asteroid at a safe distance from the ship.

- x_min/y_min is the min distance the new asteroid should be from the ship along the x-/y-axis. We add variation by using the random module, but x_min+y_min will be total BUFFER.

- x_distance/y_distance is the distance from the ship for the new asteroid along the x-/y-axis. It is a randomly selected number that ensures that the new asteroid will be at least x_min/y_min distance from the ship.

- x/y is the x-/y-coordinate for the new asteroid. We calculate it by adding the ship's x/y to x_distance/y_distance. Then I make sure x/y won't put the asteroid off the screen by "wrapping it around" the screen with the modulus operator.

- Since each asteroid should be able to call a method of the Game object, each Asteroid object needs a reference to the Game object. We pass self to the parameter game, which the Asteroid constructor will use as an attribute for the game.

- Display the new level number and play the level-up sound:

```
# display level number
level_message = games.Message(value= "Level" \
+ str(self.level), size = 40, color = color.yellow,
x = games.screen.width/2,
y = games.screen.width/10,
lifetime = 3 * games.screen.fps,
is_collideable = False)
games.screen.add(level_message)

# play new level sound (except at first level)
if self.level > 1:
    self.sound.play()
```

The end() Method

- end() displays the message "Game Over" in the middle of the screen in big, red letters for about 5 seconds. After that, the game ends and the graphics screen closes:

```
def end(self):
    """ End the game. """
    # show 'Game Over' for 5 seconds
    end_message = games.Message(value =  \
    "Game Over", size = 90, color = color.red,
    x = games.screen.width/2,
    y = games.screen.height/2,
    lifetime = 5 * games.screen.fps,
    after_death = games.screen.quit,
    is_collideable = False)
    games.screen.add(end_message)
```

Adding an Asteroid Class Variable and Constant

- Add a class constant:

```
POINTS = 30
```

- The constant will act as a base value for the number of points an asteroid is worth. The actual point value will be modified according to the size of the asteroid—smaller asteroids will be worth more than larger ones.

- In order to change levels, the program needs to know when all of the asteroids on the current level are destroyed.

- Keep track of the total number of asteroids with a new class variable, total:

```
total =  0
```

Modifying Asteroid's Constructor Method

- Add a line to increment Asteroid.total In the constructor:

```
Asteroid.total += 1
```

- We want any asteroid to be able to send the Game object a message, so we give each Asteroid object a reference to the Game object:

```
def __init__(self, game, x, y, size):
```

- The game parameter accepts the Game object, and is used to create an attribute for the new Asteroid object:

```
self.game = game
```

- So, each new Asteroid object has an attribute game to refer to the game itself. Through game, an Asteroid object can call a method of the Game object, such as advance().

Modifying Asteroid's die() Method

- Decrement Asteroid.total:

```
Asteroid.total -= 1
```

- Increase the score based on Asteroid.POINTS and the size of the asteroid. Also make sure the score is flush right:

```
self.game.score.value += \
                        int(Asteroid.POINTS / self.size)
self.game.score.right = games.screen.width - 10
```

- When we create each of the 2 new asteroids, we need to pass a reference to the Game object, by modifying the call to the Asteroid constructor:

```
new_asteroid = Asteroid(game = self.game,
```

- Test Asteroid.total to see if all the asteroids have been destroyed. If so, the final asteroid invokes the Game object's advance(), which advances the game to the next level and creates a new group of asteroids:

```
if Asteroid.total == 0:
    self.game.advance()
```

Modifying Ship's Constructor Method

- Create a class constant, VELOCITY_MAX, to limit the max velocity of the player's ship:

```
VELOCITY_MAX = 3
```

- a Ship object needs to have access to the Game object so it can invoke a Game object method:

```
def __init__(self, game, x, y):
```

- The new parameter, game, is used to create an attribute for the Ship object:

```
    self.game = game
```

- Each Ship object has an attribute game that refers to the game itself. Through game, a Ship object can call a method of the Game object, like end().

Modifying Ship's update() Method

- Cap the individual velocity components of a Ship object, dx and dy, using the class constant MAX_VELOCITY:

```
self.dx = min(max(self.dx,
-Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
self.dy = min(max(self.dy,
-Ship.VELOCITY_MAX), Ship.VELOCITY_MAX)
```

- The code ensures that dx and dy > –Ship.VELOCITY_MAX and < Ship.VELOCITY_MAX.

- **min()** returns the min of 2 numbers, while **max()** returns the max of 2 numbers.

- Cap the ship's speed to avoid several potential problems, including the ship running into its own missiles.

Adding Ship's die() Method

- When the player's ship is destroyed, the game is over:

```
def die(self):
    """ Destroy ship and end the game. """
    self.game.end()
    super(Ship, self).die()
```

The main() Function

- Create a Game object and invoke the object's play() to put the game in action:

```
def main():
    astrocrash = Game()
    astrocrash.play()

# kick it off!
main()
```