Chapter 12

Graphics:
The Pizza Panic Game

The pygame and livewires Packages

- pygame and livewires are sets of modules (called *packages*) that give Python programmers access to multimedia classes.

- With these classes, you can create programs with graphics, sound effects, music, animation. The packages allow input from a variety of devices, including the mouse and keyboard.

- With these packages, you won’t have to worry about the low-level hardware details. You can concentrate on the logic and get to writing games fast.

- pygame let you write impressive multimedia programs in Python. livewires takes advantage of the power of pygame while reducing the complexity for the programmer. livewires provides a simpler way to get started programming games with graphics and sound. And even though you won’t directly access pygame, it will work hard behind the scenes.

Installing Pygame and livewires

- Download pygame-1.9.4-cp37-cp37m-win32.whl from

http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame

- Follow the instructions in the video clip

https://www.youtube.com/watch?v=ki_5uS4bOgQ

with Python 3.7 and pygame-1.9.4-cp37-cp37m-win32.whl

- Download livewires.zip from moodle and unzip it in your Python/Scripts directory, then run **setup.py**

- Alternative: **pip install superwires**

Pygame and superwires in anaconda

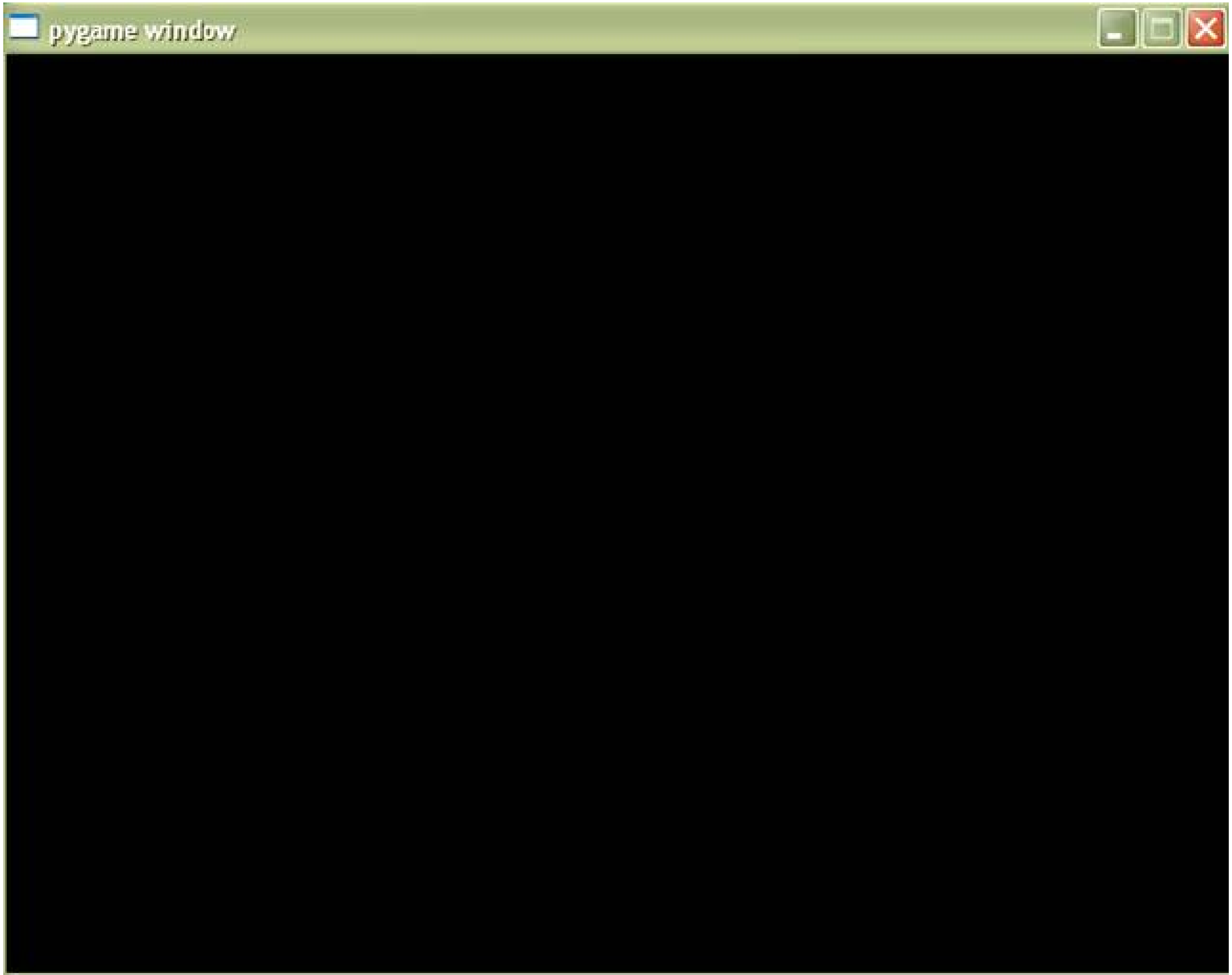
- In Anaconda, it becomes easier. Go to anaconda’s prompt

```
pip install pygame

pip install superwires
```

The New Graphics Window Program

The batch file: new_graphics_window.bat

```
new_graphics_window.py
pause
```



new_graphics_window.py

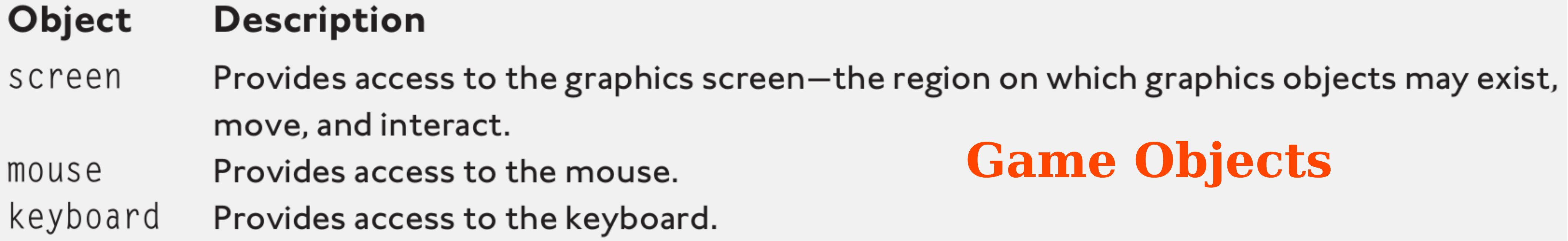
```
# New Graphics Window
# Demonstrates creating a graphics window

from superwires import games

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

games.screen.mainloop()
```

Importing the games Module

- superwires is made up of several modules, including games, containing objects & classes for game programming.

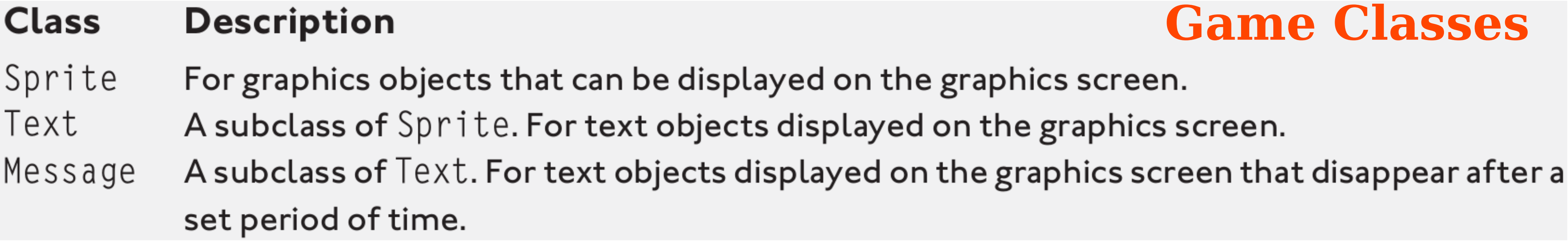
- You can import a specific module of a package by using the from statement. To import a module, use from, then a package name, then import, then a module name (or a list of module names separated by commas).

from superwires import games



Object	Description
screen	Provides access to the graphics screen—the region on which graphics objects may exist, move, and interact.
mouse	Provides access to the mouse.
keyboard	Provides access to the keyboard.

Game Objects



Game Classes

Class	Description
Sprite	For graphics objects that can be displayed on the graphics screen.
Text	A subclass of Sprite. For text objects displayed on the graphics screen.
Message	A subclass of Text. For text objects displayed on the graphics screen that disappear after a set period of time.

Initializing the Graphics Screen

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

- Call **games.init()** to create a new graphics screen.

- **screen_width** is the width of the screen, **screen_height** is the height of the screen. **fps** (short for “frames per second”) is the number of times of updating the screen every second.

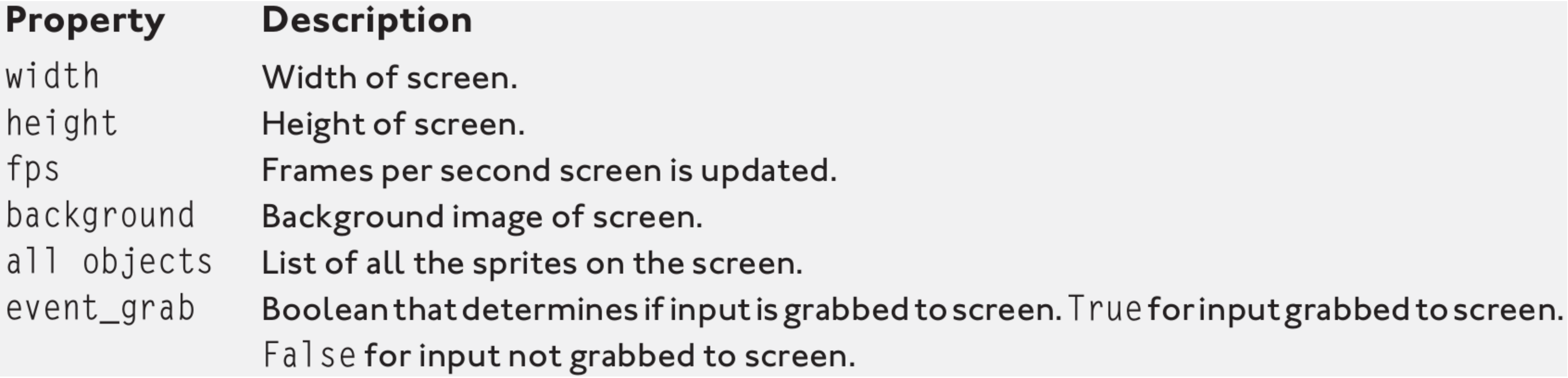
Starting the Main Loop

games.screen.mainloop()

- **screen** is the **games** object that represents the graphics screen. **mainloop()** is the workhorse of screen and updates the graphics window, redrawing everything fps times/second.

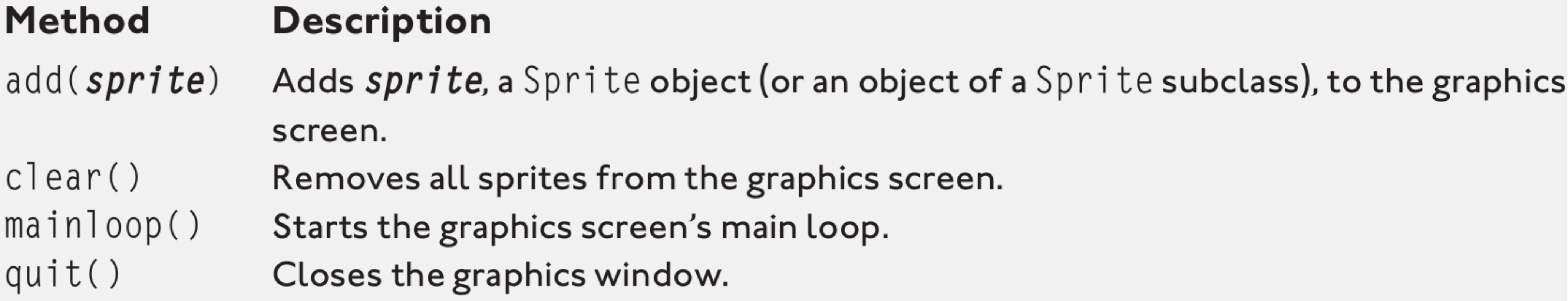
- So this line keeps the graphics window open and updates the screen 50 times per second.

- **screen**'s property:



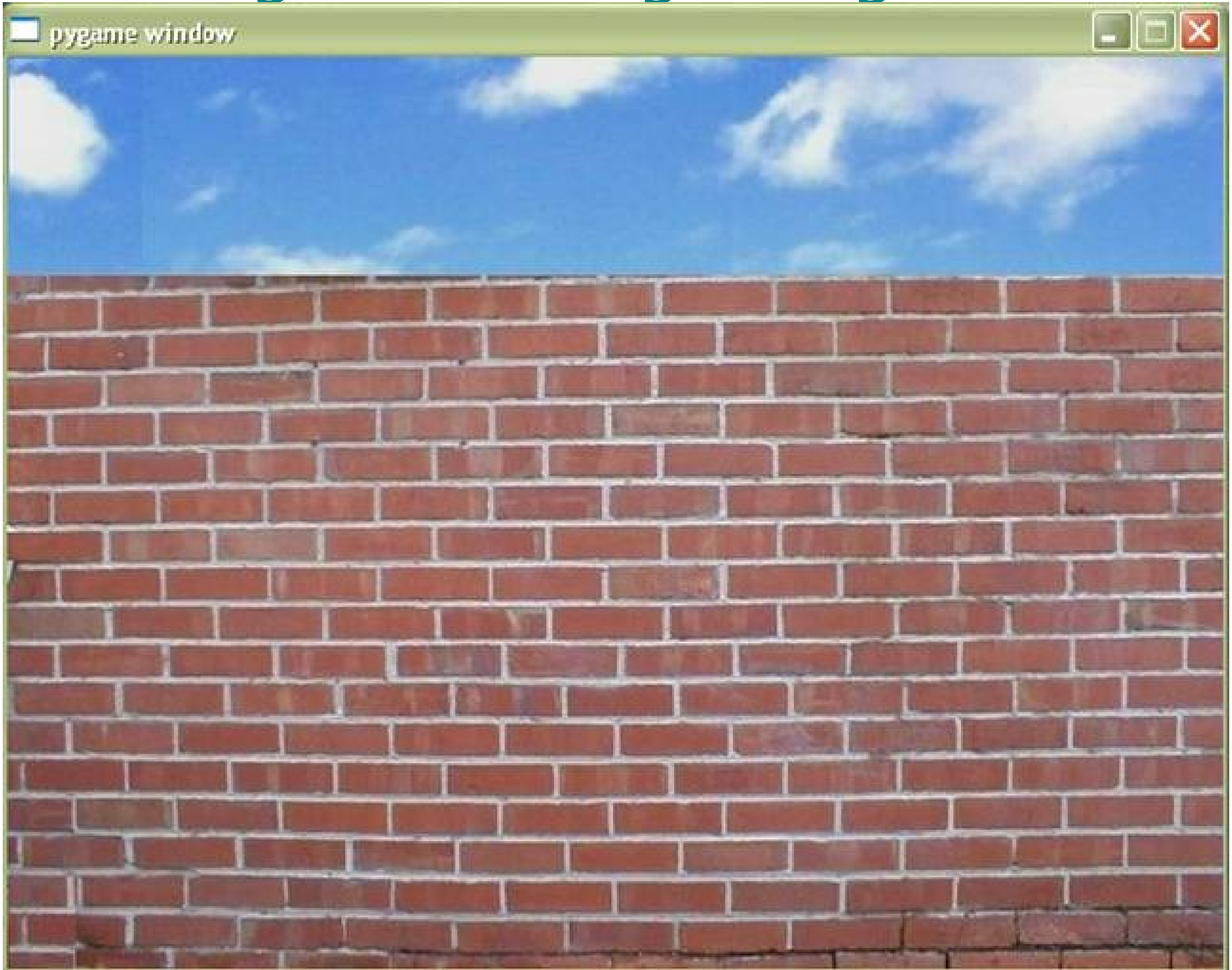
Property	Description
width	Width of screen.
height	Height of screen.
fps	Frames per second screen is updated.
background	Background image of screen.
all objects	List of all the sprites on the screen.
event_grab	Boolean that determines if input is grabbed to screen. True for input grabbed to screen. False for input not grabbed to screen.

- screen‘s methods:



Method	Description
`add(`***`sprite`***`)`	Adds ***`sprite`***, a `Sprite` object (or an object of a `Sprite` subclass), to the graphics screen.
`clear()`	Removes all sprites from the graphics screen.
`mainloop()`	Starts the graphics screen’s main loop.
`quit()`	Closes the graphics window.

The Background Image Program



background_image.py

```
# Background Image
# Demonstrates setting the background image of a
# graphics screen

from superwires import games

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

wall_image = games.load_image("wall.jpg",
                              transparent = False)
games.screen.background = wall_image

games.screen.mainloop()
```

The batch file: background_image.bat

```
background_image.py
pause
```

Loading an Image

- Before you can use an image, you have to load the image into memory to create an image object.

```
wall_image = games.load_image("wall.jpg",
                                         transparent = False)
```

- **games.load_image()** loads the image, wall.jpg, into memory and assigns it to wall_image.

- **load_image()** takes 2 arguments: a string for the file name of the image and True or False for **transparent**.

- Always load a background image with **transparent**=False.

- **load_image()** works with many image file types, including JPEG, BMP, GIF, PNG, PCX, and TGA.

Setting the Background

```
games.screen.background = wall_image
```

sets the background of the screen to wall_image.

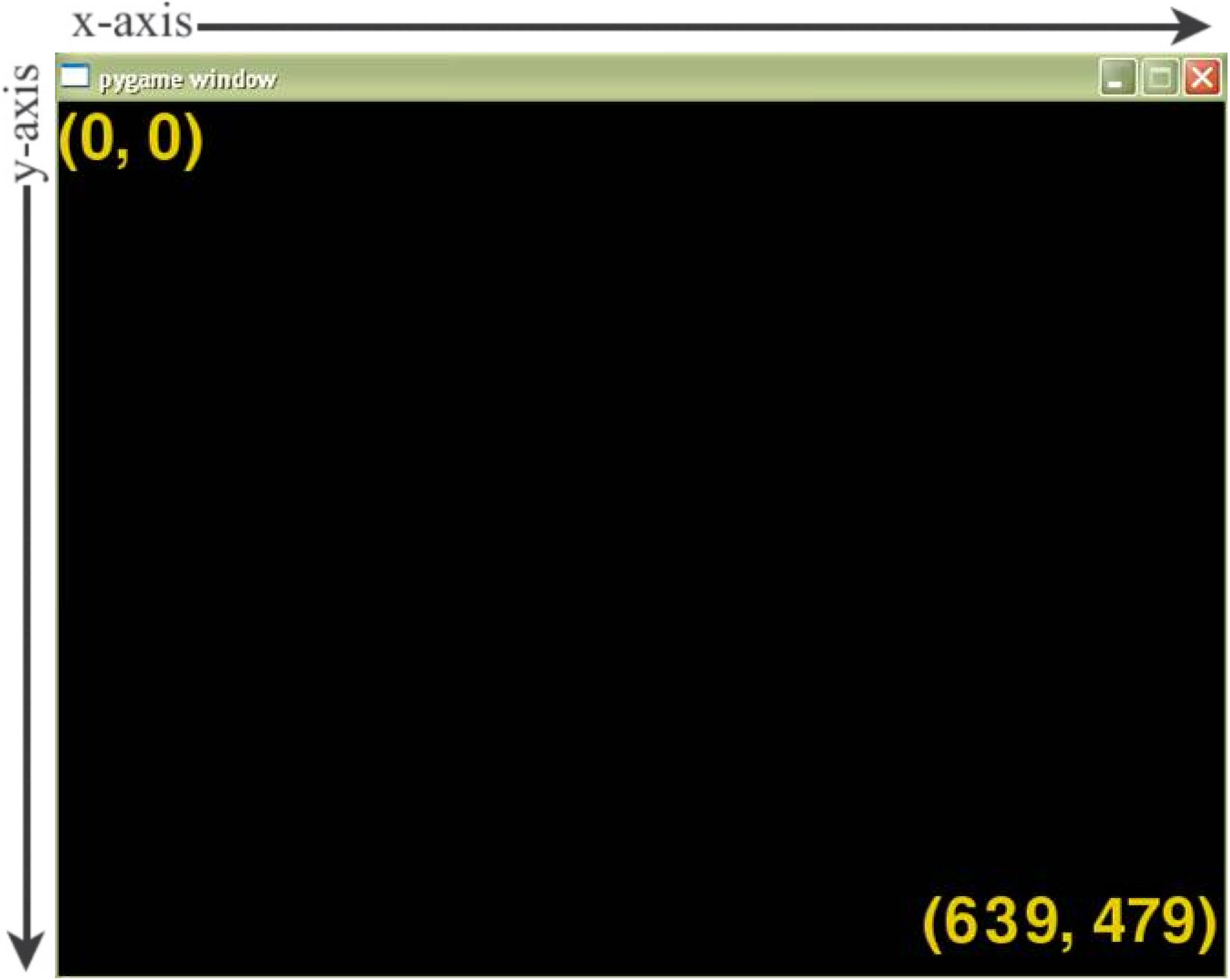
The Graphics Coordinate System

- Think of a graphics screen as a grid, 640 columns across by 480 rows down. Each intersection of a column and a row is a location on the screen, a pixel.

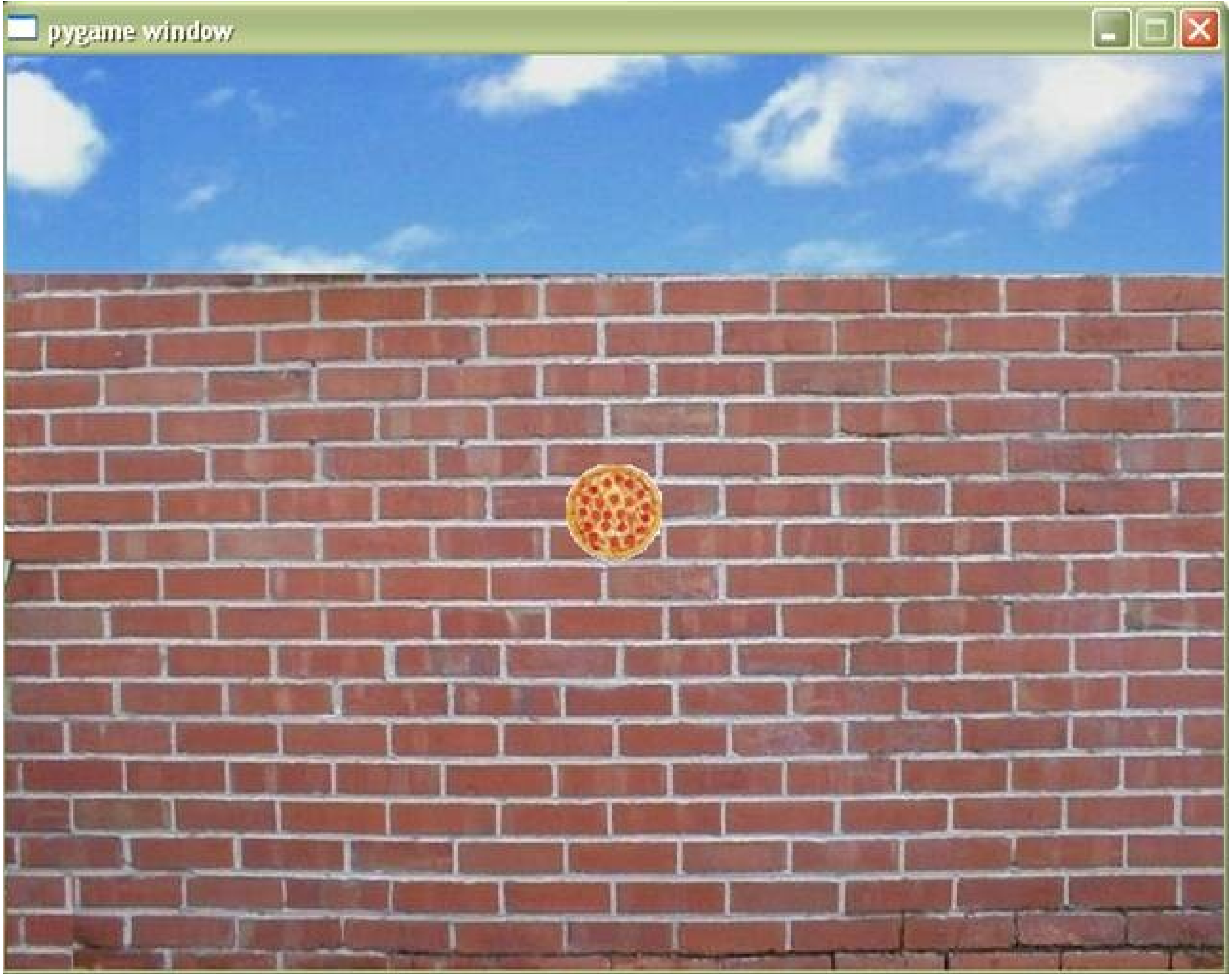
- When you talk about a specific point on the screen, you give 2 coordinates: *x* for the column, *y* for the row.

- The upper-leftmost point is (*x*,*y*)=(0,0). The point in the lower-right corner (*x*,*y*)=(639,479).

- You can place graphics objects, like the image of a pizza or the red-colored text "Game Over," on the screen using the coordinate system. The center of a graphics object is placed at the specified coordinates.


x-axis
y-axis
pygame window
(0, 0)
(639, 479)

Introducing the Pizza Sprite Program



pizza_sprite.py

- The batch file: pizza_sprite.bat

```
pizza_sprite.py
pause
```

```
# Pizza Sprite
# Demonstrates creating a sprite

from superwires import games

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

wall_image = games.load_image("wall.jpg",
                                      transparent = False)
games.screen.background = wall_image

pizza_image = games.load_image("pizza.bmp")
pizza = games.Sprite(image = pizza_image,
                                 x = 320, y = 240)
games.screen.add(pizza)

games.screen.mainloop()
```

Loading an Image for a Sprite

- Load a pizza image into memory to create an image object:

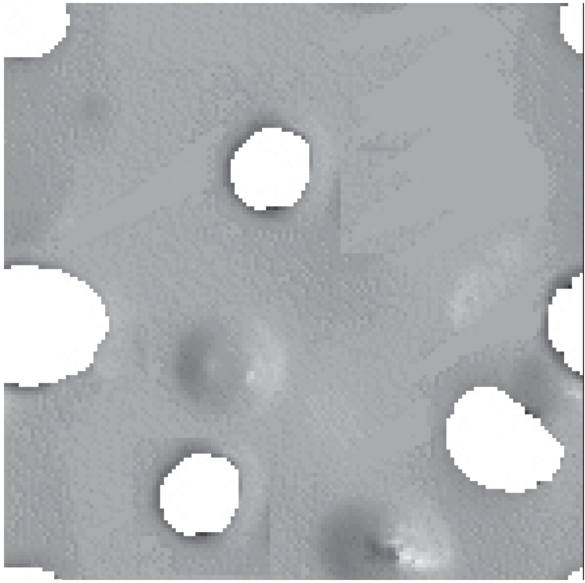
```
pizza_image = games.load_image("pizza.bmp")
```

- There is one difference in the way we load a background image, ie, we didn’t include transparent. The default value is True, so the image is loaded with transparency on.

- When an image is with transparency on, it’s displayed on a graphics screen that the background image shows through its transparent parts.

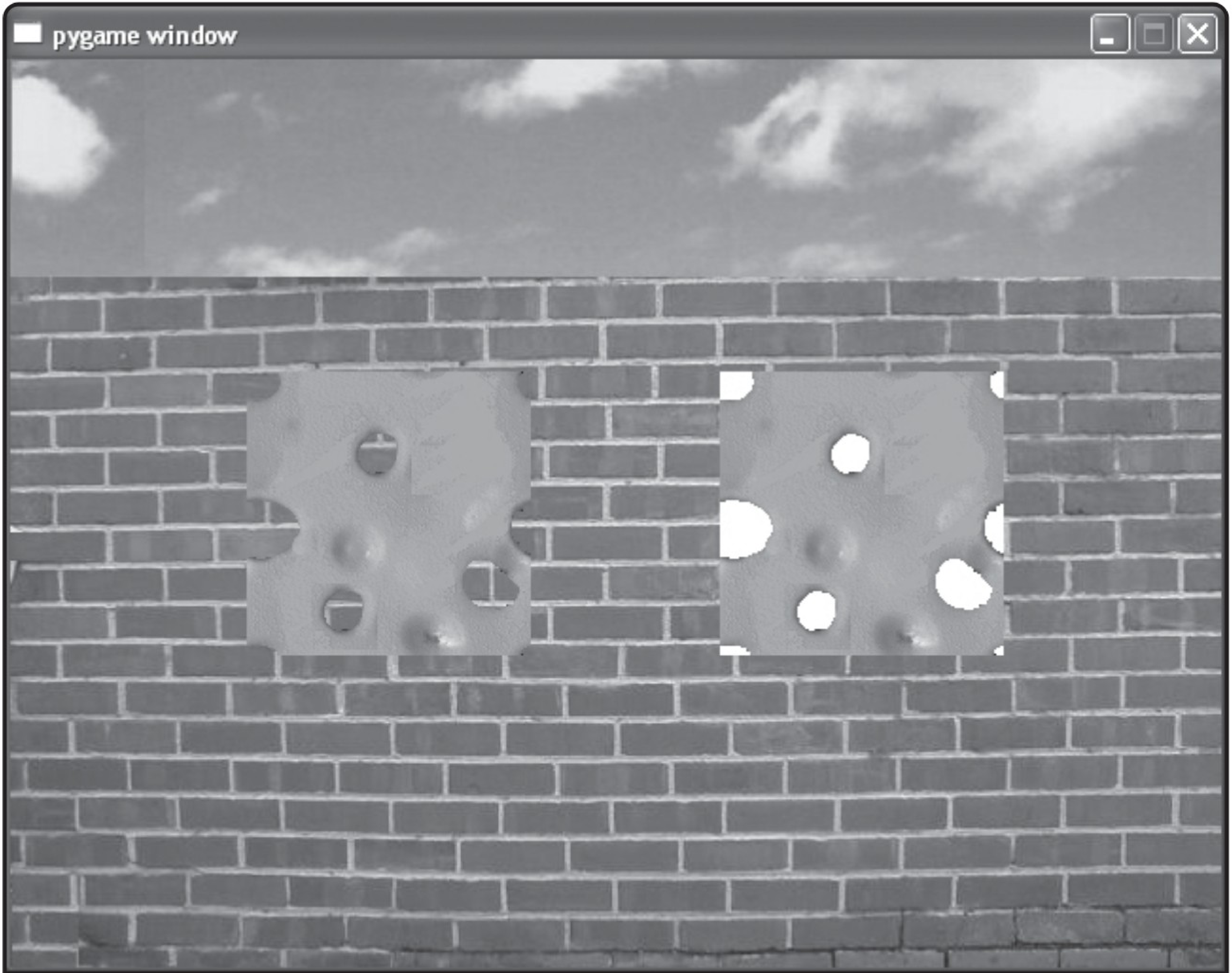
- The transparent parts of an image are defined by their color. If an image is with transparency on, the color of the point at the upper-left corner of the image is its transparent color. All parts of the image that are this transparent color will allow the background of the screen to show through.

- If we load this Swiss cheese image with transparency on, every part that is pure white (the color taken from the pixel in the image's upper-left corner) will be transparent when the sprite is displayed on a graphics screen.



- As a general rule, you'll want to create your sprite image on a solid color that is not used in any other part of the image.

- Make sure your sprite image doesn't also contain the color you're using for transparency. Otherwise, those parts of the sprite will become transparent too, making your sprite look like it has small holes or tears in it as the background image of the graphics screen shows through.


pygame window

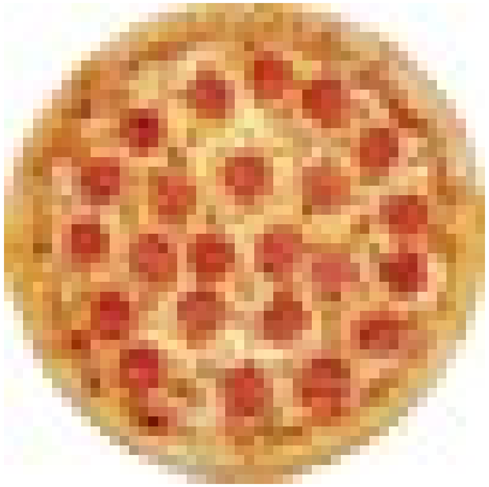
Creating a Sprite

- Create a pizza sprite:

```
pizza = games.Sprite(image = pizza_image, x = 320,
                                                    y = 240)
```

- A new **Sprite** object, pizza, is created with the image of a pizza and *x*- and *y*-coordinates of (320,240), which puts it right in the middle of the screen.

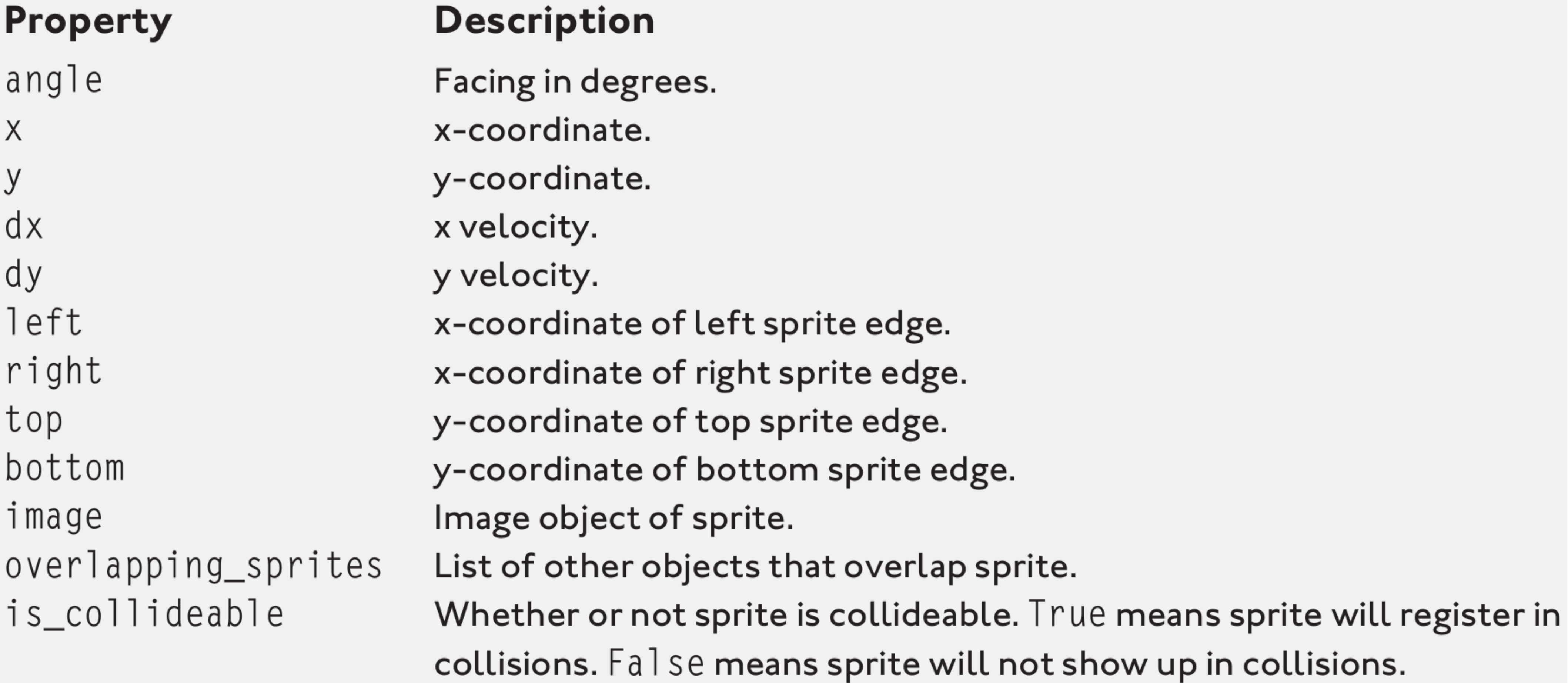
- When you create a **Sprite** object, you should pass an image, an *x*-coordinate, a *y*-coordinate to the class constructor.



Adding a Sprite to the Screen

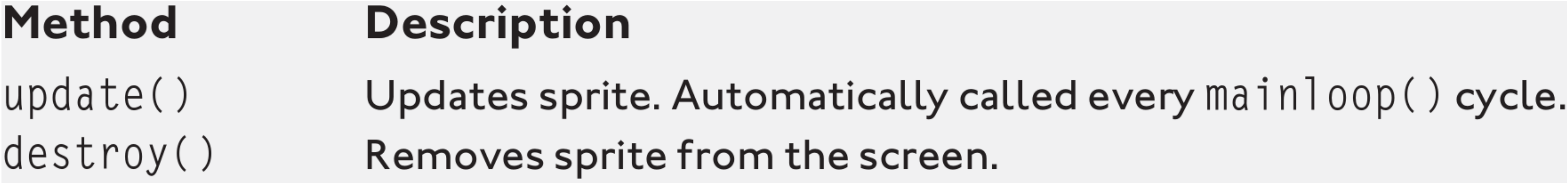
```
games.screen.add(pizza)
```

- **add()** simply adds a sprite to the graphics screen.
- Useful Sprite properties



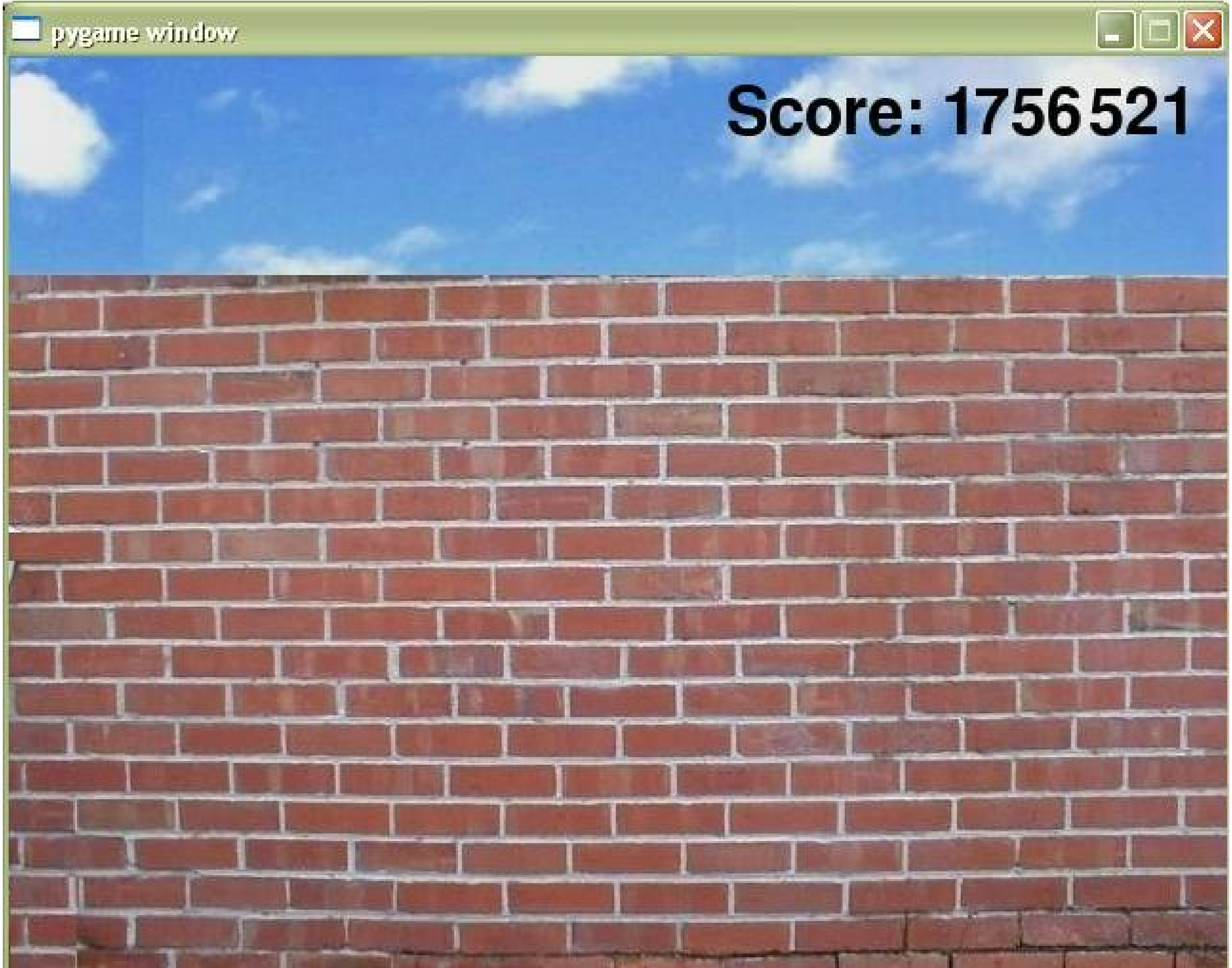
Property	Description
angle	Facing in degrees.
x	x-coordinate.
y	y-coordinate.
dx	x velocity.
dy	y velocity.
left	x-coordinate of left sprite edge.
right	x-coordinate of right sprite edge.
top	y-coordinate of top sprite edge.
bottom	y-coordinate of bottom sprite edge.
image	Image object of sprite.
overlapping_sprites	List of other objects that overlap sprite.
is_collideable	Whether or not sprite is collideable. True means sprite will register in collisions. False means sprite will not show up in collisions.

- Useful Sprite methods



Method	Description
`update()`	Updates sprite. Automatically called every `mainloop()` cycle.
`destroy()`	Removes sprite from the screen.

Introducing the Big Score Program



Score: 1756521

big_score.py

The batch file: big_score.bat

```
big_score.py
pause
```

```
# Big Score
# Demonstrates displaying text on a graphics screen

from superwires import games, color

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

wall_image = games.load_image("wall.jpg",
                                 transparent = False)
games.screen.background = wall_image

Score=games.Text(value =”Score: 1756521”, size=50,
           color = color.black, x = 500, y = 30)
games.screen.add(score)

games.screen.mainloop()
```

Importing the color Module

- livewires/superwires contains another module, **color**, which defines a set of constants that represent different colors.

- These colors can be used with certain graphics objects, including any Text or Message object.

- See the livewires/superwires documentation in Appendix B for a complete list of the predefined colors.

- To choose from a group of possible colors, we import the **color** module:

```
from superwires import games, color
```

Creating a Text Object

- A **Text** object represents text on the graphics screen.

- Create a **Text** object and assign it to score:

```
score = games.Text(value =”Score: 1756521”, size =50,
                                  color = color.black, x = 500, y = 30)
```

- At a minimum, you should pass the constructor method for a **Text** object, a value to be displayed as text, a font size, a color, an *x*-coordinate, and a *y*-coordinate.

- A **Text** object will be displayed as the string representation of whatever you pass **value**.

- **size** represents the height of the text in pixels.

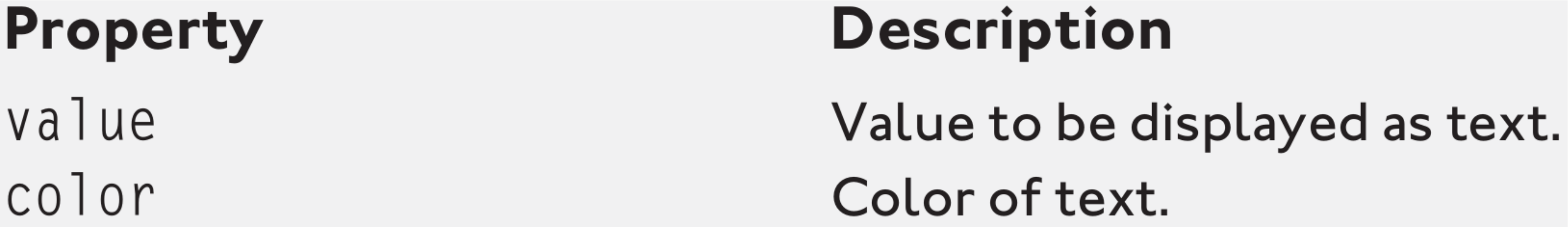
Adding a Text Object to the Screen

- add the new object to the screen so it will be displayed:

games.screen.add(score)

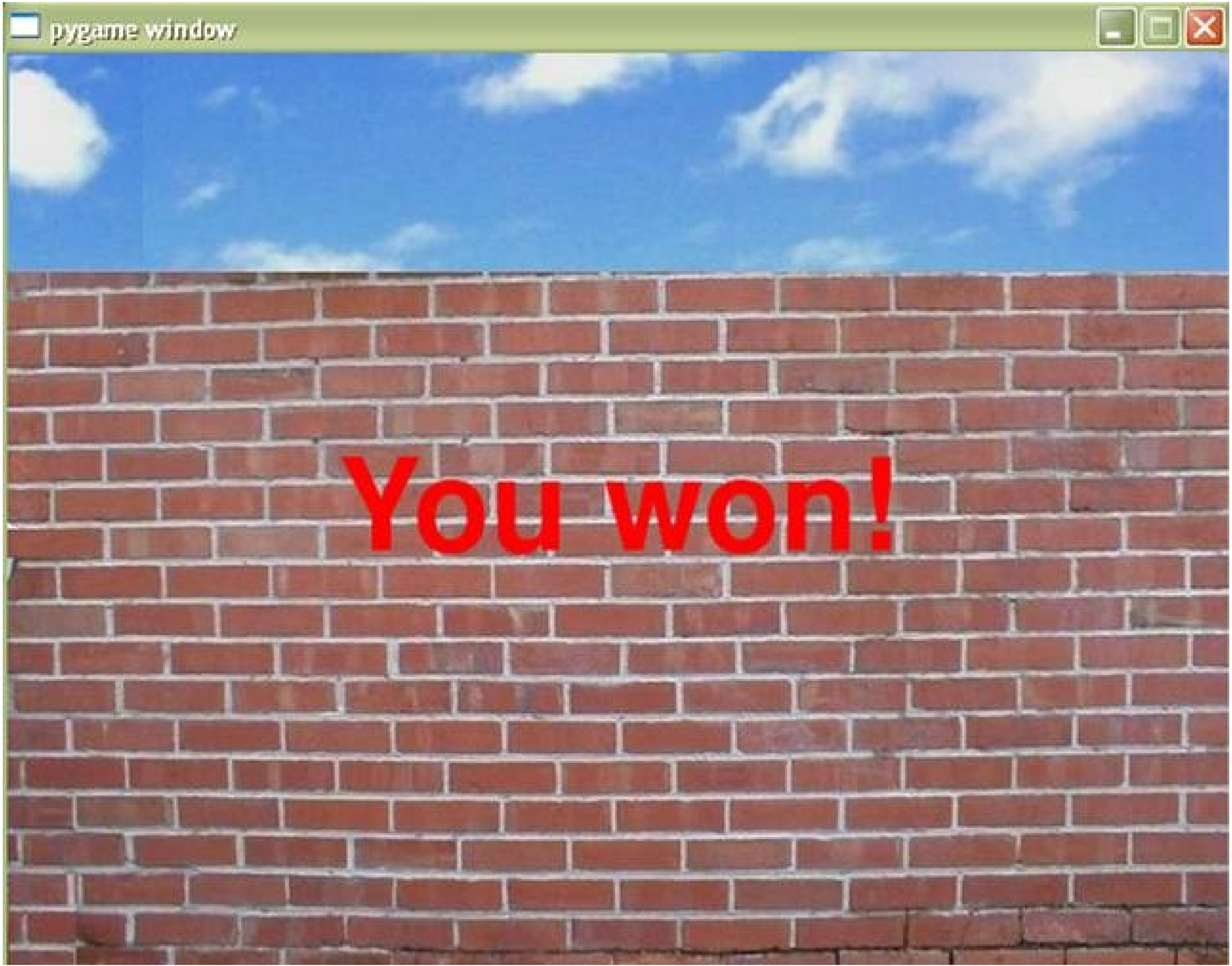
- Text is a subclass of Sprite, so Text inherits all of Sprite's properties, attributes, and methods.

- 2 additional Text properties that the class defines:



Property	Description
`value`	Value to be displayed as text.
`color`	Color of text.

Introducing the You Won Program



you_won.py

The batch file: you_won.bat

```
you_won.py
pause
```

```
# You Won
# Demonstrates displaying a message

from superwires import games, color

games.init(screen_width = 640, screen_height = 480,
                                          fps = 50)
wall_image = games.load_image("wall.jpg",
                              transparent = False)
games.screen.background = wall_image

won_message = games.Message(value = "You won!",
                    size = 100, color = color.red,
  x=games.screen.width/2, y=games.screen.height/2,
     lifetime = 250, after_death = games.screen.quit)
games.screen.add(won_message)

games.screen.mainloop()
```

Creating a Message Object

- Messages are created from the games class **Message**. A message is a special kind of Text object that destroys itself after a set period of time. A message can specify a method or a function to be executed after the object destroys itself.

- The constructor method for **Message** takes all of the values you saw with Text, but adds 2 more: **lifetime** and **after_death**. **lifetime** takes an integer value that represents how long the message is displayed, measured in mainloop() cycles. **after_death** can be passed a method or function to be executed after the **Message** object destroys itself. The default value for **after_death** is None.

- Our code instantiates a new **Message** with **lifetime** = 250. So the object lives for about 5 seconds, because mainloop() runs at 50 frames per second.

```
won_message = games.Message(value = "You won!",
                            size = 100, color = color.red,
    x=games.screen.width/2, y=games.screen.height/2,
        lifetime = 250, after_death = games.screen.quit)
```

- After the 5 seconds, games.screen.**quit()** is called, since we pass that method to **after_death**. At that point, the screen and all of its associated objects are destroyed and the program ends.

Using the Screen’s Width and Height

- The screen object has a **width** property, which represents the width of the graphics screen, a **height** property, which represents the height of the graphics screen.

- We pass values for the location of the new Message object, with x = games.screen.**width**/2, y=games.screen.**height**/2. By setting the *x*-coordinate to half of the screen width and the *y*-coordinate to half of the screen height, we put the object right in the middle of the screen.

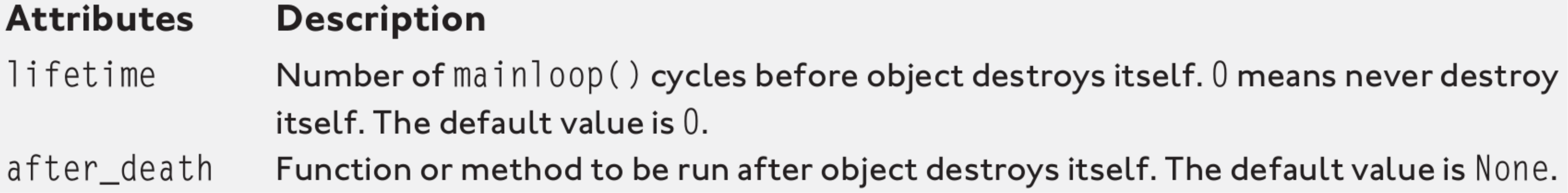
- You can use this technique to put an object in the middle of the graphics screen, independent of the actual screen width and height.

Adding a Message Object to the Screen

games.screen.add(won_message)

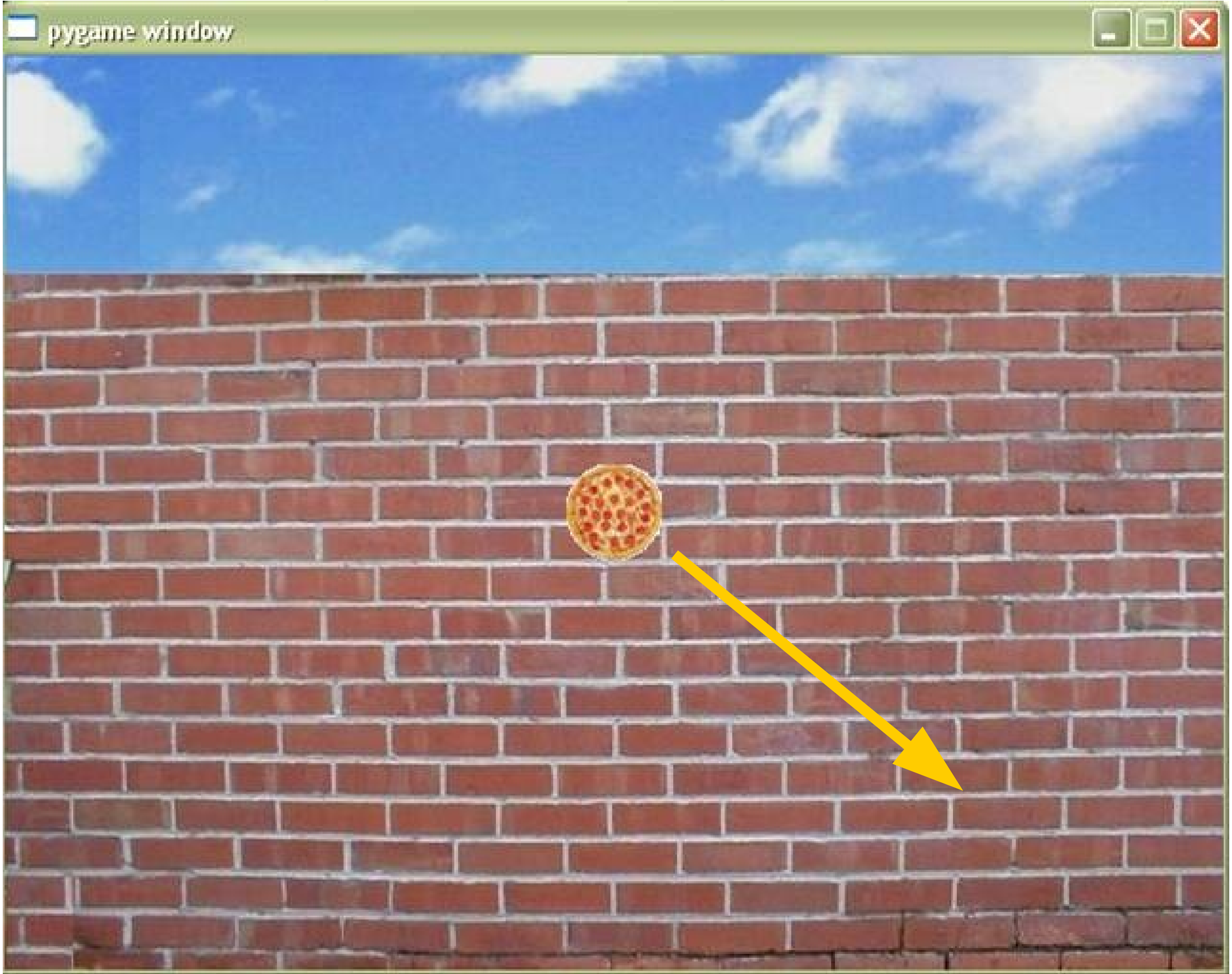
- Message is a subclass of Text. So Message inherits all of Text's properties, attributes, and methods.

- 2 additional Message attributes:



Attributes	Description
`lifetime`	Number of `mainloop()` cycles before object destroys itself. `0` means never destroy itself. The default value is `0`.
`after_death`	Function or method to be run after object destroys itself. The default value is `None`.

Introducing the Moving Pizza Program



moving_pizza.py

The batch file: moving_pizza.bat

```
moving_pizza.py
pause
```

```python
# Moving Pizza
# Demonstrates sprite velocities

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                             fps = 50)
wall_image = games.load_image("wall.jpg",
                                   transparent = False)
games.screen.background = wall_image

pizza_image = games.load_image("pizza.bmp")
the_pizza = games.Sprite(image = pizza_image,
            x = games.screen.width/2,
            y = games.screen.height/2, dx = 1, dy = 1)
games.screen.add(the_pizza)

games.screen.mainloop()
```

Setting a Sprite’s Velocity Values

- All we have to do is modify the code that creates a new sprite by providing additional values for **dx** and **dy** to the constructor method:

```
the_pizza = games.Sprite(image = pizza_image,
                         x = games.screen.width/2,
                         y = games.screen.height/2, dx = 1, dy = 1)
```

- Every object based on Sprite has **dx** and **dy** properties for the object’s velocity along the *x* and *y* axes, respectively.

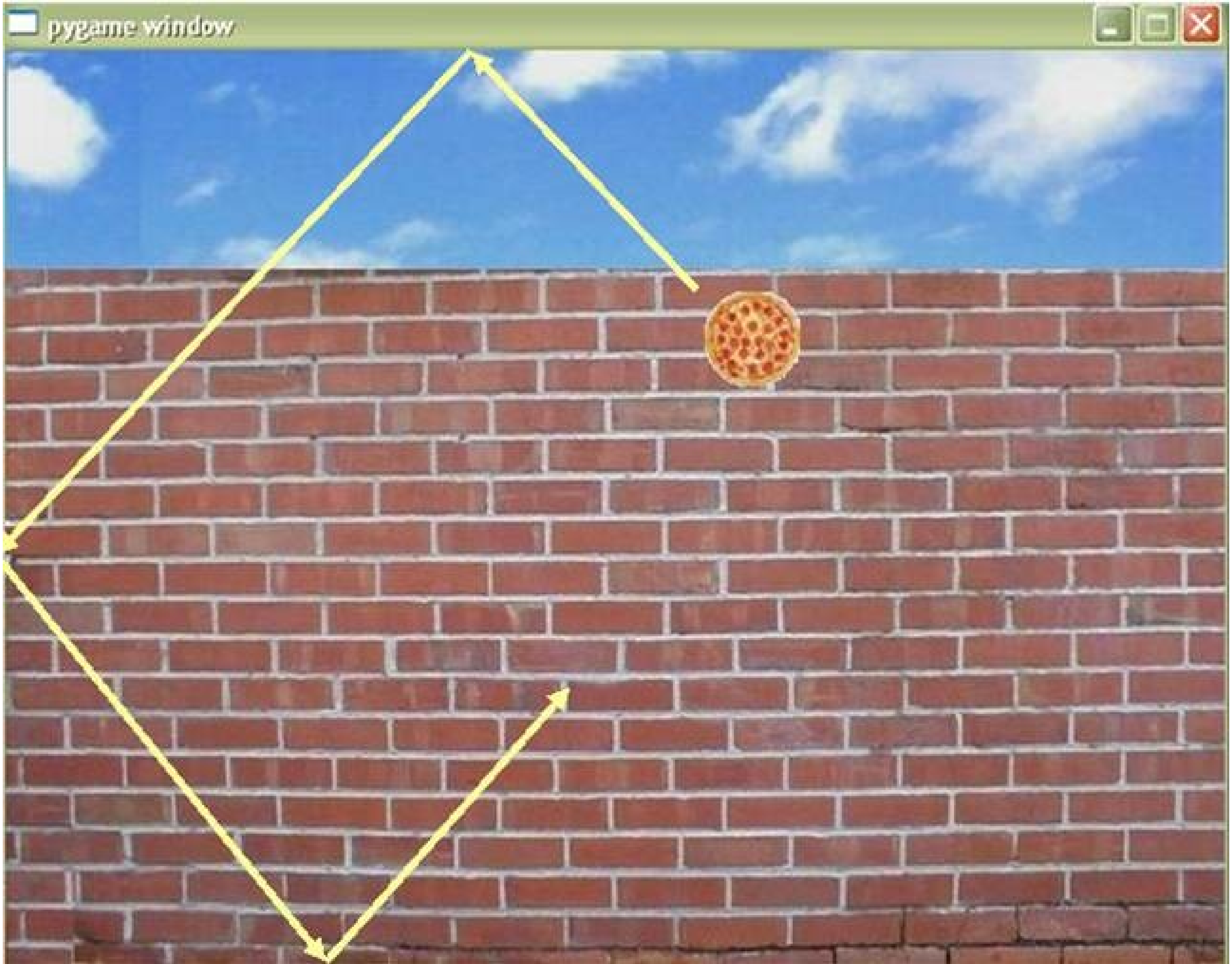
- **dx** is the change in the object’s *x*-coordinate and **dy** is the change in the object’s *y*-coordinate each time screen is updated by mainloop().

- A positive value for **dx**/**dy** moves the sprite right/down, while a negative value for **dx**/**dy** moves it left/up.

- **dx** and **dy** both have the default value of 0.

- For **dx** = 1 and **dy** = 1, every time the graphics window is updated by mainloop(), the pizza’s *x*-coordinate is increased by 1 and its *y*-coordinate is increased by 1, moving the sprite right and down.

Introducing the Bouncing Pizza Program



bouncing_pizza.py

```
# Bouncing Pizza
# Demonstrates dealing with screen boundaries

from superwires import games

games.init(screen_width = 640, screen_height = 480,
                                        fps = 50)
class Pizza(games.Sprite):
    """ A bouncing pizza. """
    def update(self):
        """ Reverse a velocity component at edge."""
        if self.right > games.screen.width or self.left < 0:
            self.dx = - self.dx

        if self.bottom>games.screen.height or self.top<0:
            self.dy = - self.dy
```

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

    pizza_image = games.load_image("pizza.bmp")
    the_pizza = Pizza(image = pizza_image,
              x = games.screen.width/2,
              y = games.screen.height/2,
              dx = 1,
              dy = 1)
    games.screen.add(the_pizza)

    games.screen.mainloop()

# kick it off!
main()
```

The batch file: bouncing_pizza.bat

```
bouncing_pizza.py
pause
```

Deriving a New Class from Sprite

- Want a sprite to do something it isn’t programmed to do: bounce. So, we need to derive a new class from Sprite:

```
class Pizza(games.Sprite):
    """ A bouncing pizza. """
```

Overriding the update() Method

- We need to add just a single method to the Pizza class to turn a moving pizza into a bouncing one.

- Every time the graphics window is updated by mainloop(), the following 2 things happen:

    - Each sprite's position is updated based on its dx and dy
    - Each sprite's **update()** method is called

- Each Sprite object has **update()**; it does nothing by default

- By overriding **update()** in Pizza, we can handle screen boundary checking.

- In the method, we check to see if the sprite is about to go beyond the screen limits in any direction. If so, we reverse the responsible velocity:

```
def update(self):
    """ Reverse a velocity component at edge."""
    if self.right > games.screen.width or self.left < 0:
        self.dx = -self.dx

    if self.bottom>games.screen.height or self.top<0:
        self.dy = -self.dy
```

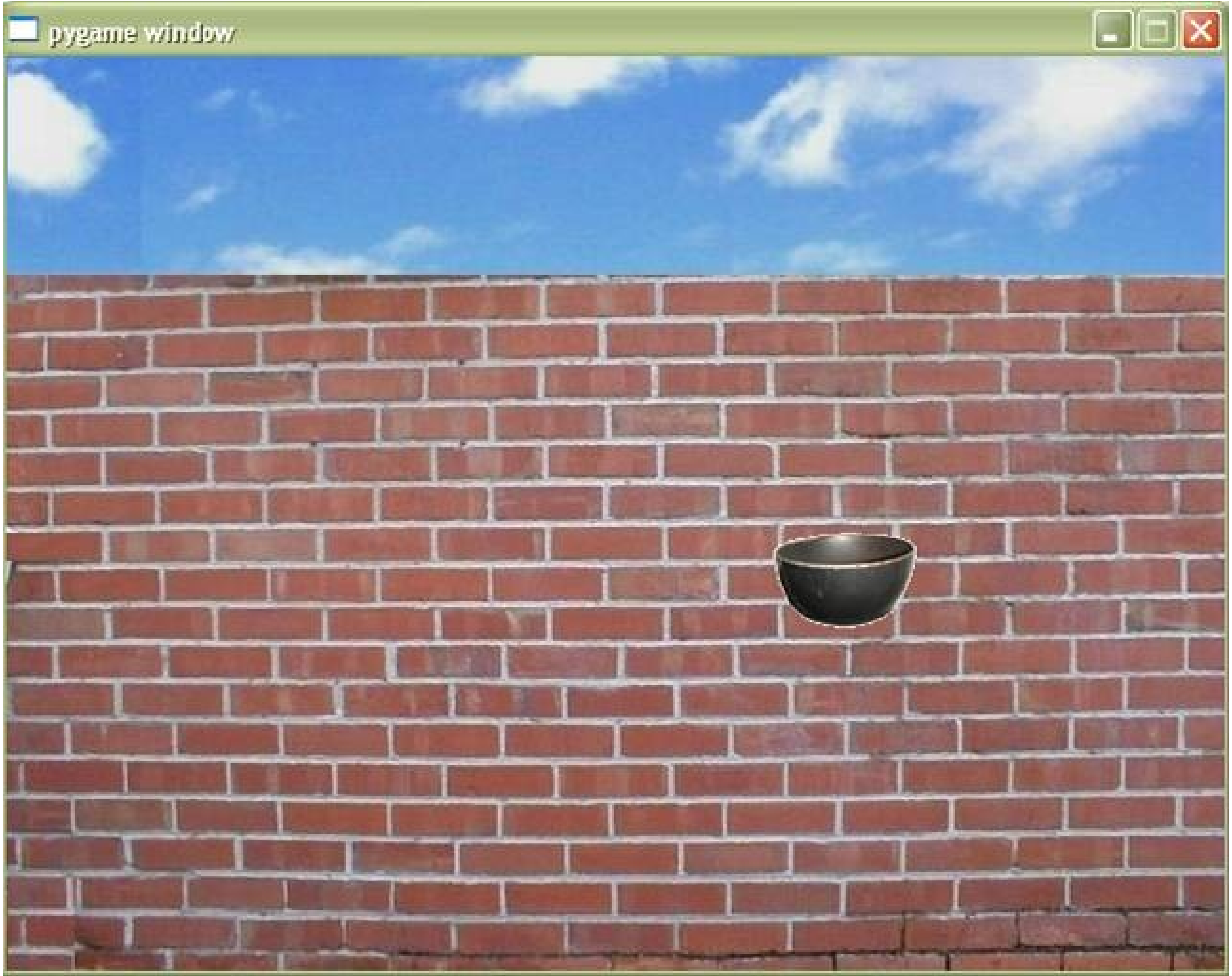
- If the object’s **right**/**bottom** property, the *x*/*y*-coordinate of its right/bottom edge, > games.screen.width/games.screen.height, or if the object’s **left**/**top** property, the *x*/*y*-coordinate of its left/top edge, < 0, then we reverse dx/dy, the pizza’s horizontal/vertical velocity, to “bounce” the pizza off the screen boundary.

Wrapping Up the Program

- We organize the rest of the code into a function main().

- One important difference is that we created an object from the new Pizza class instead of Sprite. Because of this, the object’s update() checks for screen boundaries and reverses the velocities when necessary for a pizza that bounces!

Introducing the Moving Pan Program



moving_pan.py

```
# Moving Pan
# Demonstrates mouse input

from superwires import games

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

```
class Pan(games.Sprite):
    """" A pan controlled by the mouse. """
    def update(self):
        """ Move to mouse coordinates. """
        self.x = games.mouse.x
        self.y = games.mouse.y
```

The batch file: moving_pan.bat

```
moving_pan.py
pause
```

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

    pan_image = games.load_image("pan.bmp")
    the_pan = Pan(image = pan_image,
                  x = games.mouse.x, y = games.mouse.y)
    games.screen.add(the_pan)

    games.mouse.is_visible = False

    games.screen.event_grab = True

    games.screen.mainloop()

# kick it off!
main()
```

Reading Mouse x- and y-coordinates

- Create Pan for the pan sprite:

```
class Pan(games.Sprite):
    def update(self):
        self.x = games.mouse.x
        self.y = games.mouse.y
```

- The **mouse** object has an x/y property for its *x*/*y*-coordinate . With them, we can read the current mouse location.

- In update() we assign the Pan object’s x/y the value of the **mouse** object’s x/y. It moves the pan to the current location of the mouse pointer.

- We then write a main() function that contains the type of code you’ve seen before that sets the background image and creates sprite objects:

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

    pan_image = games.load_image("pan.bmp")
    the_pan = Pan(image = pan_image,
                  x = games.mouse.x,
                  y = games.mouse.y)
    games.screen.add(the_pan)
```

- By passing games.**mouse**.x to x and games.**mouse**.y to y, the Pan object starts off at the mouse coordinates.

Setting Mouse Pointer Visibility

- use the mouse object’s **is_visible** property to set the visibility of the mouse pointer:

```
games.mouse.is_visible = False
```

- Setting the property to True means the mouse pointer will be visible, while setting it to False means the pointer will not be visible.

Grabbing Input to the Graphics Window

- Use the screen object's **event_grab** property to grab all of the input to the graphics screen:

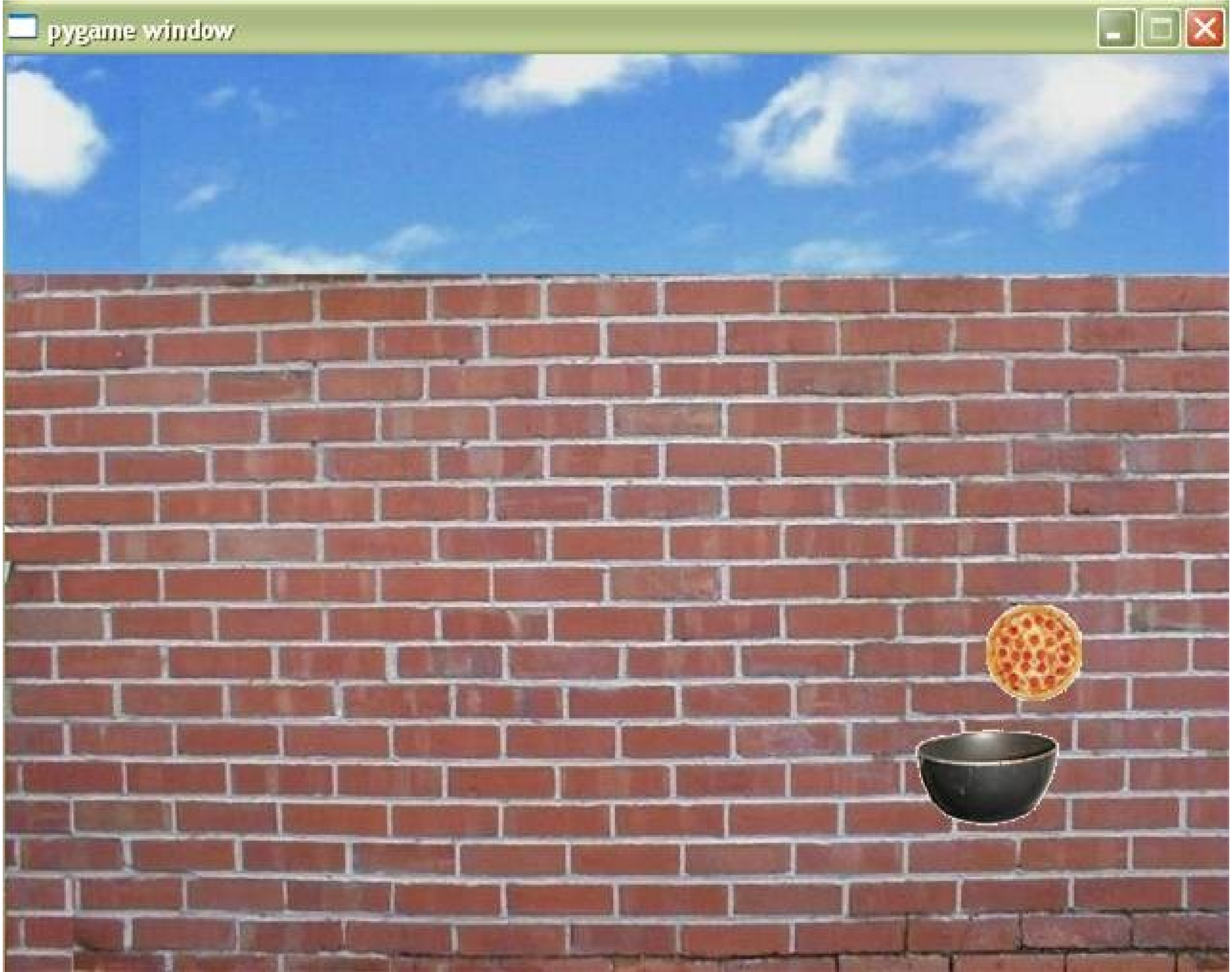
```
games.screen.event_grab = True
```

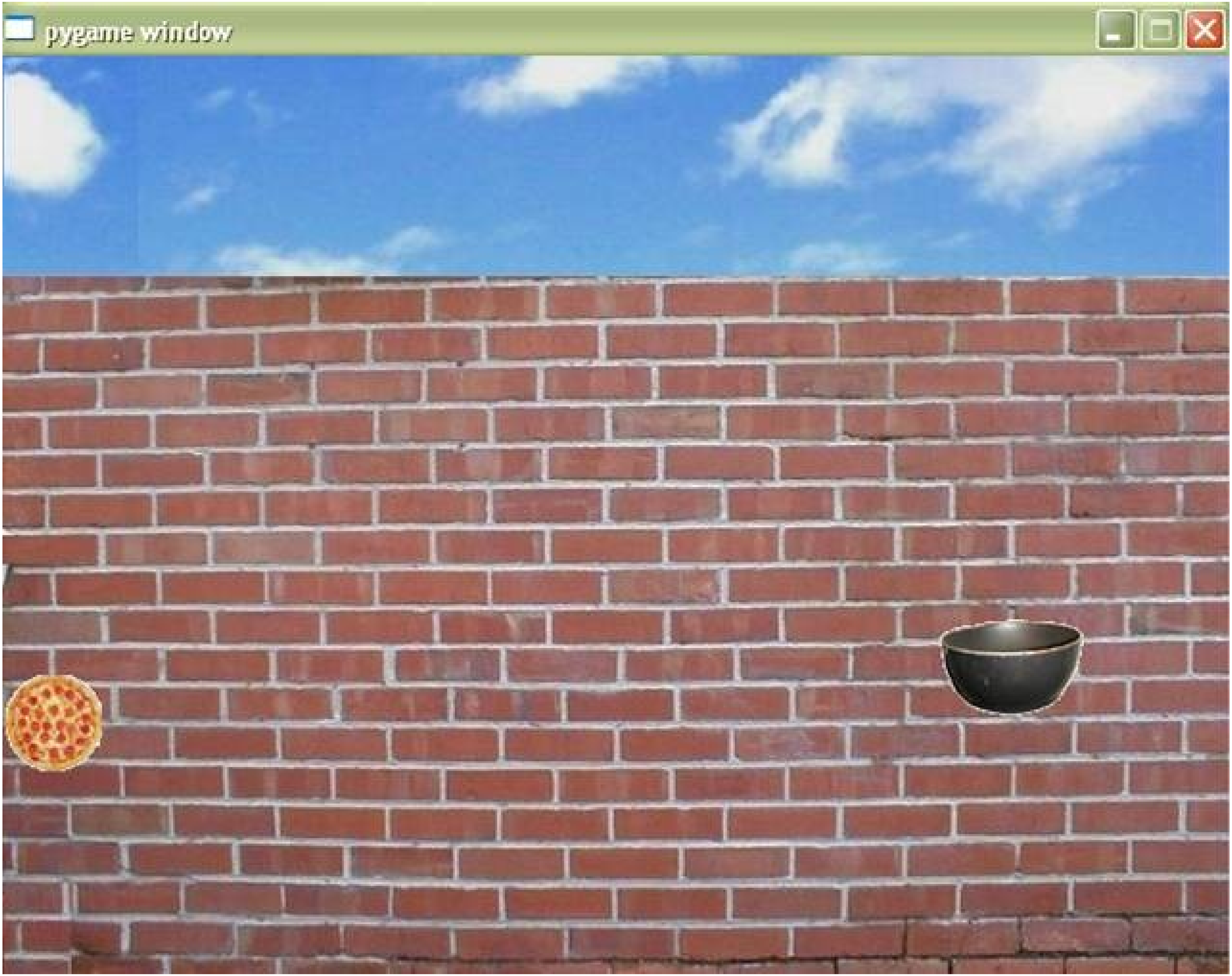
- Setting it to True means that all input will be focused on the graphics screen. The benefit of this is that the mouse won't leave the graphics window.

- Setting the it to False means that all input is not focused on the graphics screen and that the mouse pointer can leave the graphics window.

- If you grab all of the input to the graphics screen, you won't be able to close the graphics window with the mouse. However, you can always close the window by pressing the Escape key.

Introducing the Slippery Pizza Program




pygame window

slippery_pizza.py

```
# Slippery Pizza Program
# Demonstrates testing for sprite collisions

from superwires import games
import random

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

class Pan(games.Sprite):
    """" A pan controlled by the mouse. """
    def update(self):
        """ Move to mouse position. """
        self.x = games.mouse.x
        self.y = games.mouse.y
        self.check_collide()
    def check_collide(self):
        for pizza in self.overlapping_sprites:
            pizza.handle_collide()

class Pizza(games.Sprite):
    """" A slippery pizza. """
    def handle_collide(self):
        """ Move to a random screen location. """
        self.x = random.randrange(games.screen.width)
        self.y = random.randrange(games.screen.height)

def main():
    wall_image = games.load_image("wall.jpg",
                                                      transparent = False)
    games.screen.background = wall_image
    pizza_image = games.load_image("pizza.bmp")
    pizza_x = random.randrange(games.screen.width)
    pizza_y = random.randrange(games.screen.height)
    the_pizza = Pizza(image = pizza_image,
                                 x = pizza_x, y = pizza_y)
    games.screen.add(the_pizza)
    pan_image = games.load_image("pan.bmp")
    the_pan = Pan(image = pan_image,
                  x = games.mouse.x,
                  y = games.mouse.y)
    games.screen.add(the_pan)

    games.mouse.is_visible = False

    games.screen.event_grab = True

    games.screen.mainloop()

# kick it off!
main()
```

The batch file: slippery_pizza.bat

```
slippery_pizza.py
pause
```

Detecting Collisions

- Create a new Pan class by adding some code for the collision detection:

```
class Pan(games.Sprite):
    def update(self):
        self.x = games.mouse.x
        self.y = games.mouse.y
        self.check_collide()

    def check_collide(self):
        for pizza in self.overlapping_sprites:
            pizza.handle_collide()
```

- check_collide() loops through the Pan object’s **overlapping_sprites** property—a list of all of the objects that overlap it.

- Each object that overlaps the pan calls its handle_collide(). Basically, the pan tells any object that overlaps it to handle the collision.

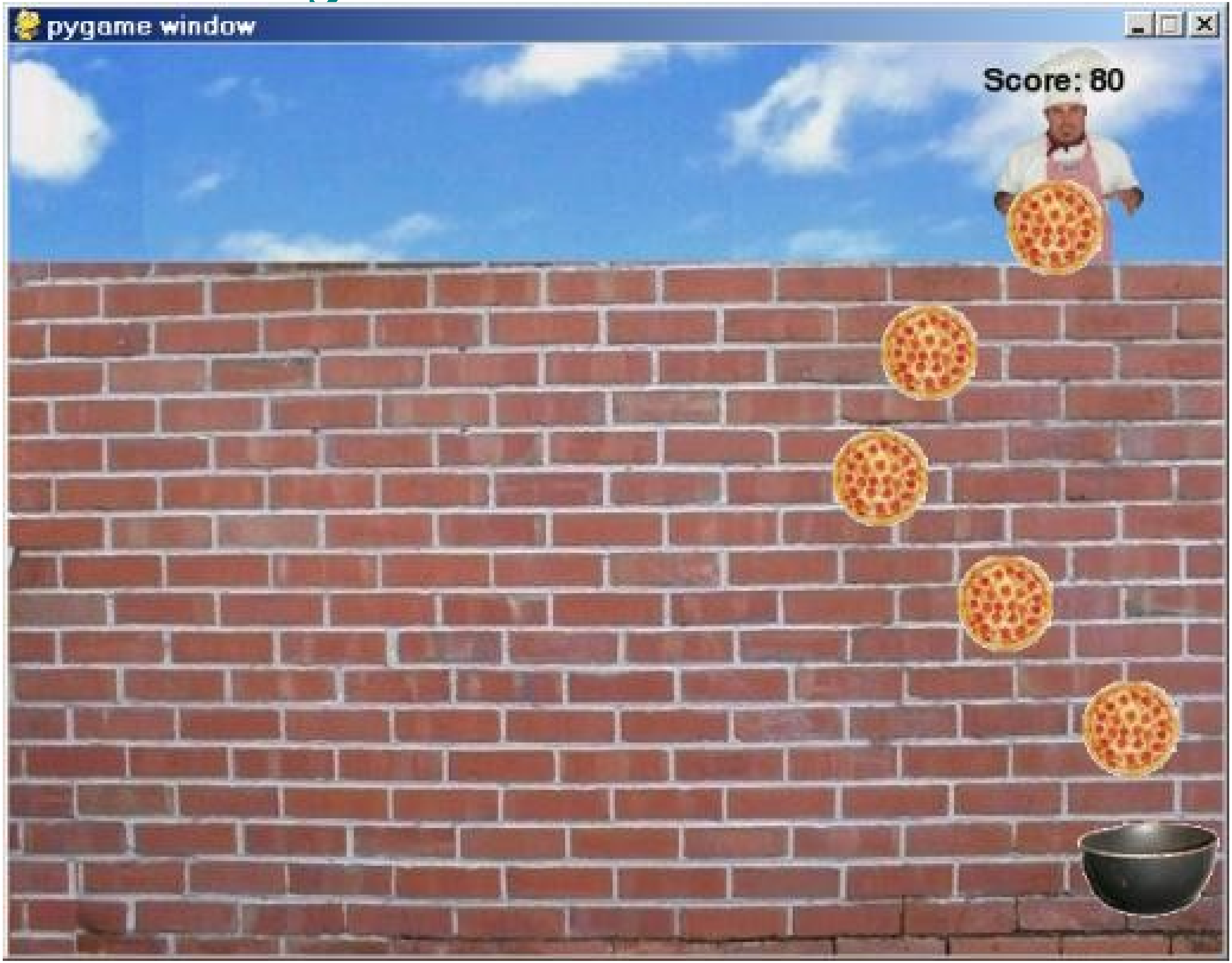
Handling Collisions

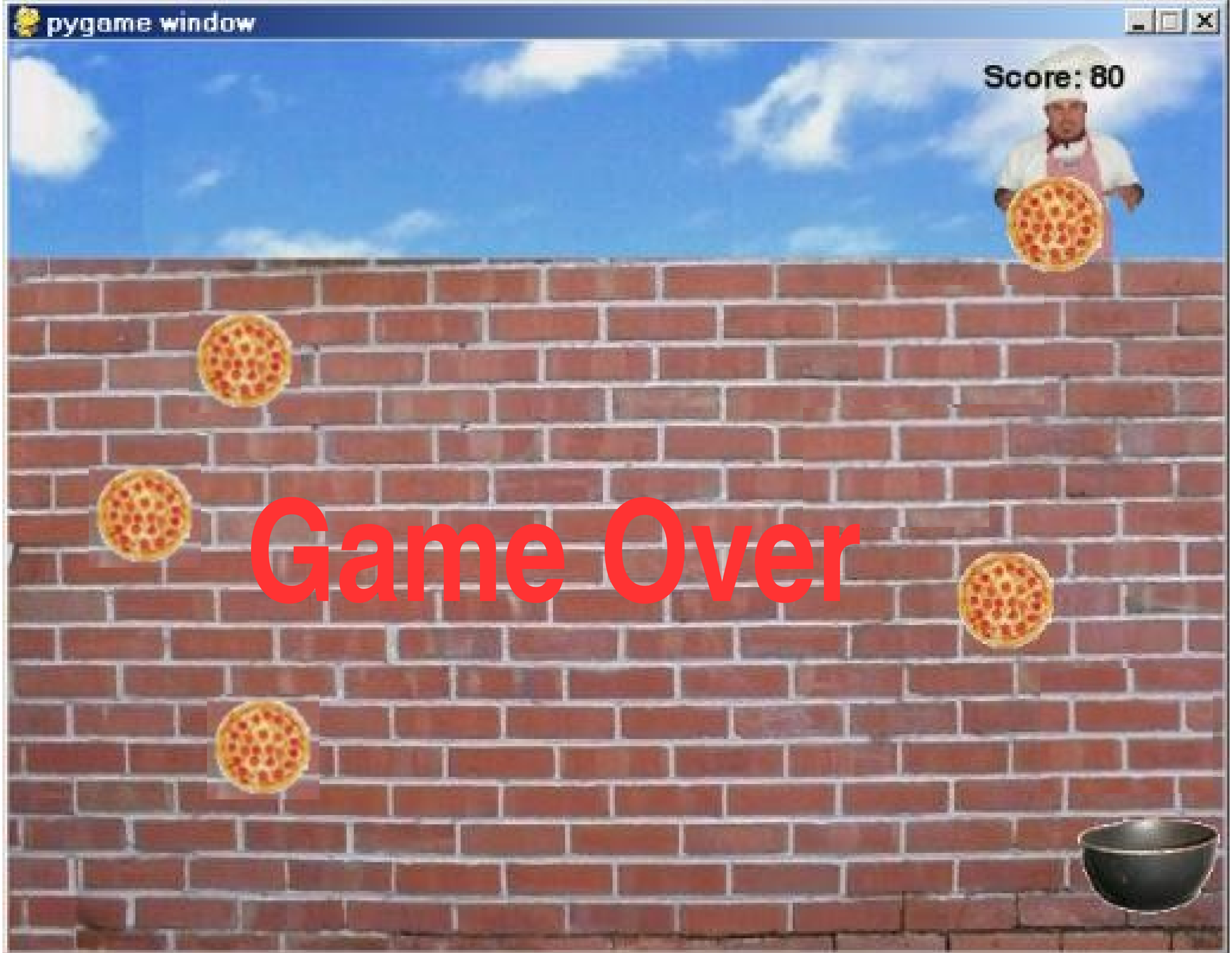
- Create a new Pizza class:

```
class Pizza(games.Sprite):
    def handle_collide(self):
        self.x = random.randrange(games.screen.width)
        self.y = random.randrange(games.screen.height)
```

- handle_collide() generates random screen coordinates and moves the Pizza object to this new location.

Introducing the Pizza Panic Game




pygame window
Score: 80
Game Over

pizza_panic.py

```
# Pizza Panic
# Player must catch pizzas before they hit the ground

from superwires import games, color
import random

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

class Pan(games.Sprite):
    """A pan controlled by player to catch pizzas. """
    image = games.load_image("pan.bmp")

    def __init__(self):
        """ Initialize Pan object and Text for score. """
        super(Pan, self).__init__(image = Pan.image,
                                  x = games.mouse.x,
                                  bottom = games.screen.height)
        self.score = games.Text(value = 0, size = 25,
                                   color = color.black, top = 5,
                                   right = games.screen.width - 10)
        games.screen.add(self.score)

    def update(self):
        self.x = games.mouse.x           # Move to mouse’s x.

        if self.left < 0:
            self.left = 0

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

        self.check_catch()

    def check_catch(self):
        for pizza in self.overlapping_sprites:   # if catched
            self.score.value += 10
            self.score.right = games.screen.width - 10
            pizza.handle_caught()
```

```
class Pizza(games.Sprite):
    """ A pizza which falls to the ground. """
    image = games.load_image("pizza.bmp")
    speed = 1

    def __init__(self, x, y = 90):
        """ Initialize a Pizza object. """
        super(Pizza, self).__init__(image = Pizza.image,
                                    x = x, y = y, dy = Pizza.speed)

    def update(self):
        """ Check if bottom reached screen bottom. """
        if self.bottom > games.screen.height:
            self.end_game()
            self.destroy()

    def handle_caught(self):
        """ Destroy self if caught. """
        self.destroy()
    def end_game(self):
        """ End the game. """
        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)
        games.screen.add(end_message)

class Chef(games.Sprite):
    """A chef moves left and right, dropping pizzas. """
    image = games.load_image("chef.bmp")

    def __init__(self, y=55, speed=2, odds_change = 200):
        """ Initialize the Chef object. """
        super(Chef, self).__init__(image = Chef.image,
                 x = games.screen.width / 2, y = y, dx = speed)

        self.odds_change = odds_change
        self.time_til_drop = 0
```

```
def update(self):
    """ Decide if direction needs to be reversed. """
    if self.left < 0 or self.right > games.screen.width:
        self.dx = -self.dx
    elif random.randrange(self.odds_change) == 0:
       self.dx = -self.dx

    self.check_drop()

 def check_drop(self):
    """Decrease countdown/drop pizza & reset. """
    if self.time_til_drop > 0:
        self.time_til_drop -= 1
    else:
        new_pizza = Pizza(x = self.x)
        games.screen.add(new_pizza)

        # set buffer to approx 30% of pizza height
        self.time_til_drop = int(new_pizza.height * 1.3 /
                                             Pizza.speed) + 1
```

```
def main():
    """ Play the game. """
    wall_image = games.load_image("wall.jpg",
                                  transparent = False)
    games.screen.background = wall_image

    the_chef = Chef()
    games.screen.add(the_chef)

    the_pan = Pan()
    games.screen.add(the_pan)

    games.mouse.is_visible = False

    games.screen.event_grab = True
    games.screen.mainloop()

# start it up!
main()
```

The batch file: pizza_panic.bat

```
pizza_panic.py
pause
```

The Pan Class

- The Pan class is a blueprint for the pan sprite that the player controls with the mouse. However, the pan will only move left and right.

- Load a sprite image to a class variable, image, because Pizza Panic has several classes, and loading an image in its corresponding class definition is cleaner than loading all of the images in main():

```
class Pan(games.Sprite):
    image = games.load_image("pan.bmp")
```

The __init__() Method

- Write the constructor to initialize a new Pan object:

```
def __init__(self):
    """ Initialize Pan object and Text for score. """
    super(Pan, self).__init__(image = Pan.image,
                              x = games.mouse.x,
                              bottom = games.screen.height)

    self.score = games.Text(value = 0, size = 25,
                            color = color.black, top = 5,
                            right = games.screen.width - 10)
    games.screen.add(self.score)
```

- We use super() to make sure that Sprite.init() is called. And we define an attribute score —a Text object—for the player's score, which begins at 0.

The update() Method

- update() moves the player’s pan:

```
def update(self):
    self.x = games.mouse.x

    if self.left < 0:
        self.left = 0

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

    self.check_catch()
```

- update() assigns the mouse *x*-coordinate to the Pan object’s *x*-coordinate, allowing the player to move the pan left and right with the mouse.

- Use the object’s left/right to check if its left/right edge is less/greater than 0/games.screen.width —meaning that part of the pan is beyond the left/right edge of the graphics window.

- If it is, we set the left/right edge to 0/games.screen.width so that the pan is displayed at the left/right edge of the window.

The check_catch() Method

- check_catch() checks if the player has caught any of the falling pizzas:

```
def check_catch(self):
    for pizza in self.overlapping_sprites:
        self.score.value += 10
        self.score.right = games.screen.width - 10
        pizza.handle_caught()
```

- For each object that overlaps the pan, check_catch() increases the player’s score by 10.

- Then it ensures that the right edge of the Text object for the score is always 10 pixels from the right edge of the screen, no matter how many digits long the score gets.

The Pizza Class

- This class is for the falling pizzas that the player must catch:

```
class Pizza(games.Sprite):
    image = games.load_image("pizza.bmp")
    speed = 1
```

- image for the pizza image and speed for the pizzas’ falling speed.

- We set speed to 1 so that the pizzas fall at a slow speed.

The __init__() Method

- __init__() initializes a new Pizza object:

```
def __init__(self, x, y = 90):
    super(Pizza, self).__init__(image = Pizza.image,
                                x = x, y = y, dy = Pizza.speed)
```

- In this method we call the constructor of the super class of Pizza.

- We set the default value for y to 90, which puts each new pizza right at the chef’s chest level.

The update() Method

- update() handles screen boundary checking:

```
def update(self):
    if self.bottom > games.screen.height:
        self.end_game()
        self.destroy()
```

- update() checks if a pizza has reached the bottom of the screen. If it has, the method invokes the object’s end_game() and then the object removes itself from the screen.

The handle_caught() Method

- handle_caught() is invoked by the Pan object when the Pizza object collides with it:

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

- When a pizza collides with a pan, the pizza is considered “caught” and simply ceases to exist. So, the Pizza object invokes its own destroy() and the pizza literally disappears.

The end_game() Method

- end_game() ends the game. It's invoked when a pizza reaches the bottom of the screen:

```
def end_game(self):
    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)
    games.screen.add(end_message)
```

- It creates a Message object that declares game over. After ~5 seconds, the message disappears and the window closes.

- end_game() is called when a pizza reaches the bottom. Since "Game Over" lasts ~ 5 seconds, it's likely for another pizza to reach the bottom before the graphics window closes—resulting in multiple "Game Over" messages.

The Chef Class

- The Chef class is used to create the crazy chef who throws the pizzas off the restaurant rooftop:

```
class Chef(games.Sprite):
    image = games.load_image("chef.bmp")
```

- Define a class attribute, image, for the chef image.

The __init__() Method

```
def __init__(self, y=55, speed=2, odds_change = 200):
    super(Chef, self).__init__(image = Chef.image,
            x = games.screen.width / 2, y = y, dx = speed)

    self.odds_change = odds_change
    self.time_til_drop = 0
```

- Call the constructor of the super class of Chef. Pass image the class attribute Chef.image. Pass x to put the chef in the middle. y=55 puts the chef on top of the brick wall. dx is passed speed, determining the chef’s horizontal velocity as he moves along the rooftop. The default value is 2.

- odds_change represents the odds that the chef changes his direction. If odds_change=200, then there’s a 1/200 chance that every time the chef moves, he’ll reverse direction.

- time_til_drop represents the amount of time, in mainloop() cycles, until the chef drops his next pizza. We set it to 0 initially, meaning that when a Chef object springs to life, it should immediately drop a pizza.

The update() Method

```
def update(self):
    if self.left < 0 or self.right > games.screen.width:
        self.dx = -self.dx
    elif random.randrange(self.odds_change) == 0:
       self.dx = -self.dx

    self.check_drop()
```

- A chef slides along the rooftop in one direction until he either reaches the edge of the screen or “decides,” at random, to switch directions.

- update() checks if the chef has moved beyond the left or right edge. If he has, then he reverses direction with the code self.dx = - self.dx. Or the chef has a 1/odds_change chance of changing direction.

The check_drop() Method

- The method is invoked every mainloop() cycle:

```
def check_drop(self):
    if self.time_til_drop > 0:
        self.time_til_drop -= 1
    else:
        new_pizza = Pizza(x = self.x)
        games.screen.add(new_pizza)
        self.time_til_drop = int(new_pizza.height * 1.3 /
                                                  Pizza.speed) + 1
```

- time_til_drop represents a countdown. If time_til_drop > 0 , then 1 is subtracted from it. Or a new Pizza object is created and time_til_drop is reset.

- The new time_til_drop is set so that the pizza is dropped when the distance from the previous one is about 30% of the pizza height, independent of how fast the pizzas are falling.

The main() Function

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

    the_chef = Chef()
    games.screen.add(the_chef)

    the_pan = Pan()
    games.screen.add(the_pan)

    games.mouse.is_visible = False

    games.screen.event_grab = True
    games.screen.mainloop()

# start it up!
main()
```