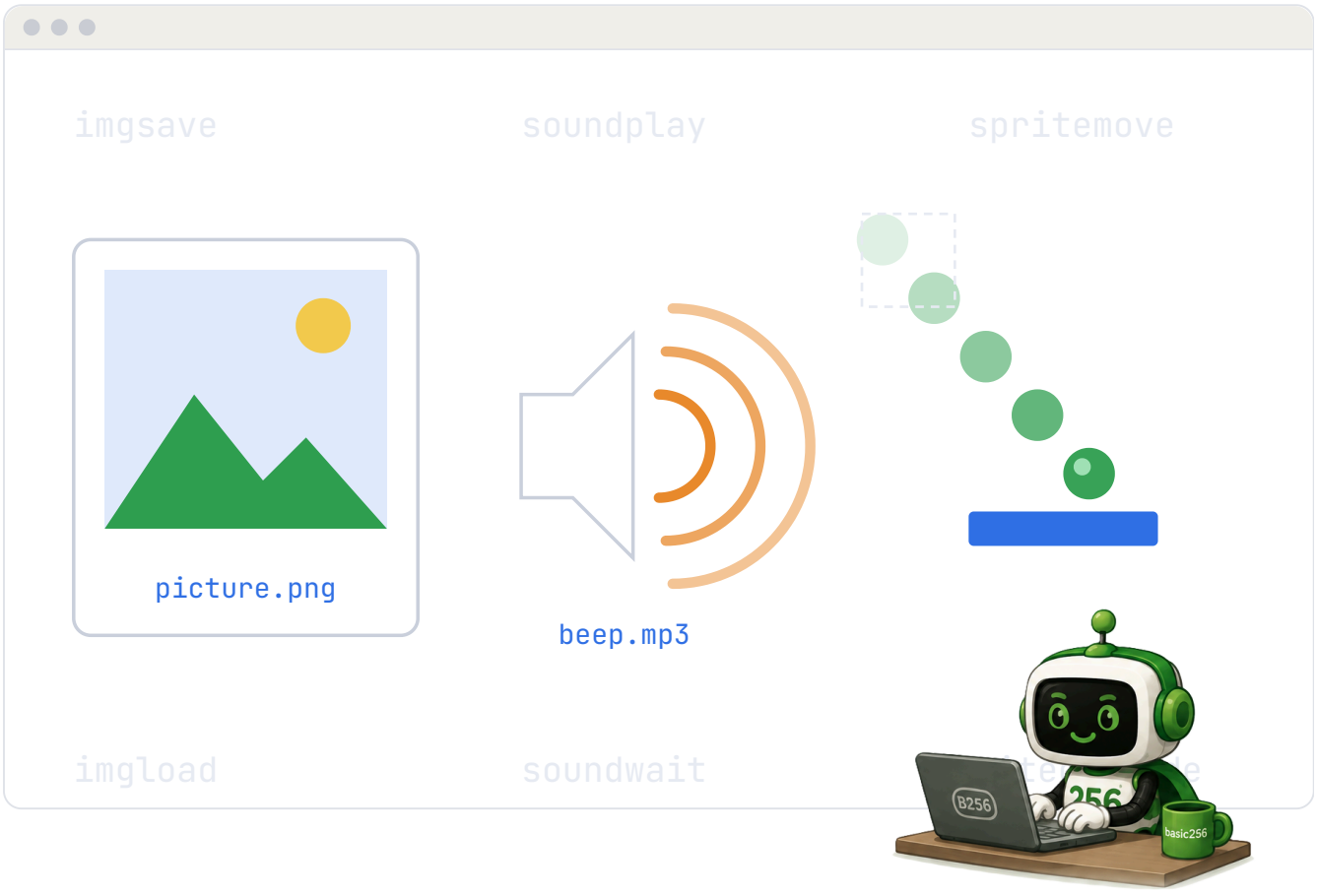
LEARN TO CODE · PICTURES, SOUNDS AND SPRITES

Chapter 13

Images, Sounds, and Sprites

So far your programs have drawn everything themselves, and the only sounds were tones and speech. In this chapter your programs save their drawings as picture files and load pictures back, scaled and turned. They play sound files, such as MP3 and WAV recordings. And they use **sprites**: pictures that move around the screen on their own layer and can tell when they bump into each other. At the end you put it all together in a paddle ball game.



© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

Saving images to a file

The `imgsave` statement turns the graphics output into a picture file.

Everything your programs draw can be saved as a picture file. You can then use it on a website, in a presentation, or in another program. Program 84 draws a series of pentagons with **stamp** (Chapter 9), each a little bigger and turned a little further, to make a geometric flower. Then **imgsave** saves it as a PNG (Portable Network Graphics) file called *5pointed.png*, in the folder where the program is saved.

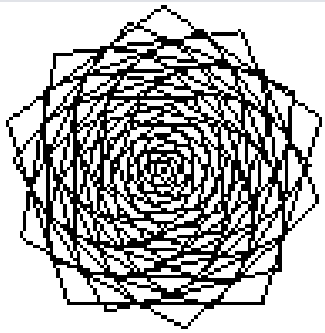
5pointed.kbs BASIC-256

```
# 5pointed.kbs - save a picture to a file

graphsize 100, 100
clg white
color black, clear
for s = 1 to 50 step 2
   stamp 50, 50, s, s, {0, -1, .95, -.31, .59, .81,
      -.59, .81, -.95, -.31}
next s

imgsave "5pointed.png", IMAGETYPE_PNG
```

GRAPHICS OUTPUT



5pointed.png, 100 × 100 pixels.

Program 84 · Save an Image

graphsize (Chapter 9) makes the graphics output only 100 × 100 pixels, so the picture file is small too: **imgsave** saves the whole graphics output, no more and no less. When the program ends, the next program starts with the usual 500 × 500 again.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
imgsave filename
imgsave filename, type
```

imgsave saves the graphics output to a picture file. Without a type, the picture is saved as a PNG file. You can give the type as a string or as a constant:

STRING	CONSTANT	
`"png"`	`IMAGETYPE_PNG`	the default; keeps the transparent parts
`"jpg"` or `"jpeg"`	`IMAGETYPE_JPG`	good for photos; the transparent parts turn black
`"bmp"`	`IMAGETYPE_BMP`	Windows bitmap; the transparent parts turn black

A file name without a folder is saved in the folder of your program. If you name a file somewhere else on the computer, BASIC-256 first asks whether the program may write there.

The programs in this chapter need a few small pictures: a green ball, a blue ball and a paddle. Program 85 draws them and saves each one in its own file. It draws on a **transparent** background: `clg clear` wipes the graphics output to nothing at all, not even white. In a PNG file those transparent pixels stay see-through, so when you load the picture later, only the ball itself covers what is underneath, not a white square around it.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

makesprites.kbs BASIC-256

```
# makesprites.kbs - draw the pictures for this chapter
# and save them as PNG files

# a green ball and a blue ball, 30 x 30 pixels
graphsize 30, 30
clg clear
call drawball(rgb(0, 150, 60), rgb(150, 230, 170))
imgsave "greenball.png"
clg clear
call drawball(rgb(0, 50, 200), rgb(150, 180, 255))
imgsave "blueball.png"

# a paddle, 80 x 16 pixels
graphsize 80, 16
clg clear
color darkblue
rect 0, 0, 80, 16
color blue
rect 2, 2, 76, 12
imgsave "paddle.png"

print "Saved greenball.png, blueball.png and paddle.png."
end

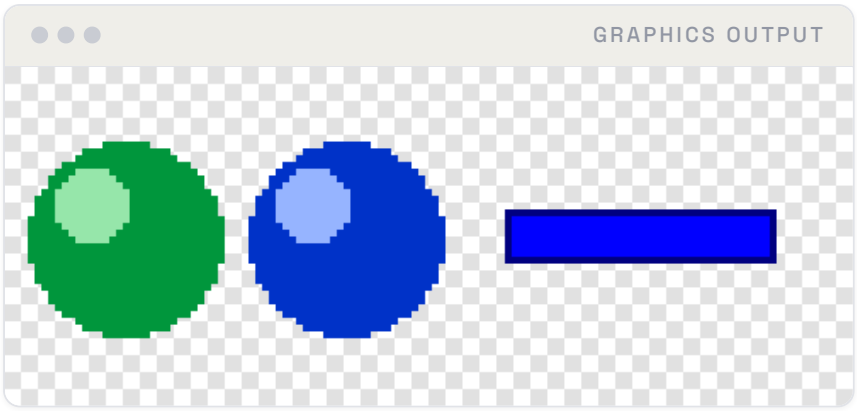
subroutine drawball(dark, light)
   color dark
   circle 15, 15, 14
   # a shine, up and to the left
   color light
   circle 10, 10, 5
end subroutine
```

Program 85 · Make the Pictures for This Chapter

TEXT OUTPUT

```
Saved greenball.png, blueball.png and paddle.png.
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



The three pictures, shown larger on a checkerboard so you can see which parts are transparent.



NEW CONCEPT

The color **clear** is transparent: it draws nothing. `clg clear` — or just **clg** — makes the whole graphics output transparent; `clg white` fills it with white. On the screen, transparent looks white, but a PNG file keeps the difference. A JPG or BMP file can't store transparency, so there the transparent parts turn black: use `clg white` before you save one of those.



TRY IT

Run Program 85, then add a third ball in your favourite colors and save it as *myball.png*. Look at the files in your program's folder.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

Images from a file

The `imgload` statement draws a picture from a file on the graphics output.

Keep your files together. When a program loads a file by its name only, BASIC-256 looks for it in the folder where the program is saved. So save your program first, in the same folder as the pictures and sounds it uses — before you run it. Programs 86 to 92 load the pictures Program 85 made, so save them all in one folder.

Files that come with BASIC-256. Some programs in this chapter also use a picture and sounds from the Examples folder that comes with BASIC-256. Choose **File › Open Example...** to see that folder, and copy these files into your own folder (you can't save your programs in the Examples folder itself):

FILE	FOLDER IN EXAMPLES	USED BY
`bibot.png`	Sprites	Program 87
`shakedice.mp3`	Original_Examples › dice	Program 88
`click.mp3, toggle.mp3`	Original_Examples › sprites	Programs 89 and 92
`beep.mp3`	Original_Examples › sprites	Program 92

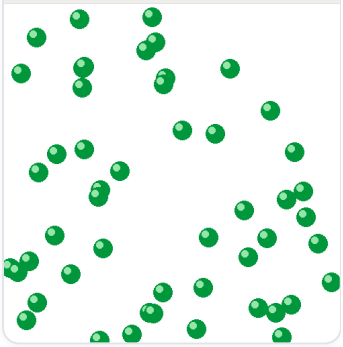
Program 86 loads *greenball.png* fifty times and draws it at random places. Notice that *x* and *y* give the **center** of the picture, not its top left corner as for **rect**. That is why some balls are cut off at the edges.

imgloadball.kbs — BASIC-256

```
# imgloadball.kbs - load a picture from a file

clg white
for i = 1 to 50
   x = rand * graphwidth
   y = rand * graphheight
   imgload x, y, "greenball.png"
next i
```

GRAPHICS OUTPUT



Program 86 · Imgload a Graphic

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
imgload x, y, filename
imgload x, y, scale, filename
imgload x, y, scale, rotation, filename
```

imgload reads a picture file and draws it on the graphics output, with its center at *x*, *y*. It can read BMP, PNG, GIF, JPG and JPEG files.

A *scale* makes the picture bigger or smaller: 1 is its own size, 2 twice as big, 0.5 half. A *rotation* turns it clockwise around its center, by an angle in radians (0 to 2π), just like **stamp** in Chapter 9. To rotate a picture without scaling it, give a scale of 1.

Program 87 uses the scale and the rotation. It draws BitBot fifty times in the middle of the screen, each time a little bigger and turned a little further, until the last BitBot — more than three times its own size and turned a full circle — sits upright on top of the others.

imgloadbitbot.kbs BASIC-256

```
# imgloadbitbot.kbs - load a picture, scaled and rotated
# bibot.png comes with BASIC-256: Examples, Sprites folder

clg white
x = graphwidth / 2
y = graphheight / 2
for i = 1 to 50
   scale = i / 16
   angle = 2 * pi * i / 50
   imgload x, y, scale, angle, "bibot.png"
next i
say "Hello BitBot."
```

GRAPHICS OUTPUT



Program 87 · Imgload a Graphic with Scaling and Rotation

The variable *scale* grows from 1/16 to 50/16, and *angle* from 2π/50 to a whole turn, 2π. At the end, **say** (Chapter 1) greets BitBot.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Playing sounds from a file

The `soundplay` statement plays a sound file while your program keeps running.

In Chapter 4 you made music with **sound**, and **say** speaks text aloud. BASIC-256 can also play recorded sounds from files, such as MP3 and WAV files. **soundplay** starts a sound file and goes on with the next statement straight away: the sound plays in the background while the program carries on.

Program 88 starts the sound of shaking dice and, while it rattles, “pops” three random numbers from 1 to 6 onto the screen, each one growing from tiny to huge, and each faster than the last. Then **soundwait** waits until the dice have stopped rattling, and **say** announces the last number.

numberpopper.kbs — BASIC-256

```
# numberpopper.kbs - popping numbers with a sound effect
# shakedice.mp3 comes with BASIC-256: Examples,
# Original_Examples, dice folder (Mike Koenig, CC BY 3.0)

fastgraphics
soundplay "shakedice.mp3"

speed = 0.05
for t = 1 to 3
   n = int(rand * 6 + 1)
   for pt = 10 to 300 step 10
      clg white
      color black
      font "Tahoma", pt, 700
      text 10, 10, n
      refresh
      pause speed
   next pt
   speed = speed / 2
next t
# wait for the sound to finish
soundwait

say "You rolled a " + n
end
```

GRAPHICS OUTPUT



The last number, at its biggest.

Program 88 · Popping Numbers with Sound Effects

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

NEW CONCEPT

```
soundplay filename
soundwait
soundstop
```



soundplay loads a sound file and starts playing it. The program doesn't wait: the next statement runs at once, while the sound plays in the background. BASIC-256 plays WAV and MP3 files. Each **soundplay** starts a new sound, so if you play another one before the first has finished, you hear both.

soundwait stops the program until the last sound you started has finished. **soundstop** stops that sound at once.

In Chapter 8 you saw **soundplay** play tones; with a file name it plays the file. **sound** from Chapter 4 also plays a file, but waits until it has finished, like **soundplay** followed by **soundwait**.

wavplay, wavwait and wavstop are old. Older programs and books use `wavplay`, `wavwait` and `wavstop`. BASIC-256 2.3 still runs them, but prints a warning that they are obsolete. Use `soundplay`, `soundwait` and `soundstop` instead. The **Advanced Sound and Music** chapter shows more: loading a sound once to play it many times, changing the volume, pausing, and playing several sounds under control.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

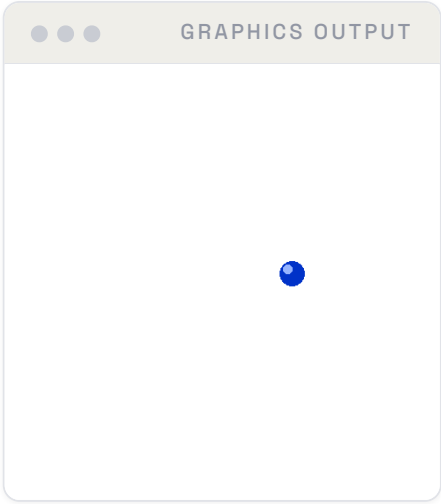
SECTION 4

Moving images — sprites

Sprites are pictures that move around the screen without redrawing everything else.

To animate a ball so far, your programs erased the screen, drew everything again at its new place, and called **refresh**. A **sprite** is a picture that lives on its own layer above the graphics output. You just move it, and BASIC-256 takes care of the rest: what is underneath stays as it was. Sprites can also tell you when one touches another (a **collision**). That makes games and animations much easier to write.

Program 89 bounces a blue ball around the screen, with a click on the sides and another sound on the top and bottom.



The program makes room for one sprite with **spritedim**, loads it from *blueball.png* with **spriteload**, puts it at 100, 100 with **spriteplace** and shows it with **spriteshow**. In the loop, **spritemove** moves it *dx* pixels to the right and *dy* down. Because the position of a sprite is its center, the ball touches a side when its center is one radius, *r*, away from it. **spritew** gives the width of the sprite, and half of that is the radius.

sprite1ball.kbs — BASIC-256

```
# sprite1ball.kbs - bounce a ball sprite, with sounds
# click.mp3 and toggle.mp3 come with BASIC-256: Examples,
# Original_Examples, sprites folder (Mike Koenig, CC BY 3.0)

clg white

spritedim 1
spriteload 0, "blueball.png"
spriteplace 0, 100, 100
spriteshow 0

dx = rand * 5 + 5
dy = rand * 5 + 5
r = spritew(0) / 2   # the radius of the ball

while true
   spritemove 0, dx, dy
   if spritex(0) <= r or spritex(0) >= graphwidth - r then
      dx = dx * -1
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
        soundplay "click.mp3"
    end if
    if spritey(0) <= r or spritey(0) >= graphheight - r then
        dy = dy * -1
        soundplay "toggle.mp3"
    end if
    pause 0.02
end while
```

Program 89 · Bounce a Ball with Sprite and Sound Effects

Compare this with the bouncing ball in Chapter 8: there is no **clg**, no drawing and no **refresh**. To use sprites, a program

1. says how many sprites it needs (**spritedim**),
2. makes each one from a picture file (**spriteload**) or from a polygon (**spritepoly**, later in this section),
3. puts it in place (**spriteplace**) and makes it visible (**spriteshow**),
4. and then moves it around (**spritemove** or **spriteplace**).

Functions tell where a sprite is (**spritex** and **spritey**), how big it is (**spritew** and **spriteh**), and whether it is visible (**spritev**).



NEW CONCEPT

```
spritedim numberofsprites
```

spritedim makes room for the given number of sprites. They are numbered from 0: after `spritedim 3` you have sprites 0, 1 and 2. You may have as many sprites as your program needs, but a lot of big sprites can slow it down.



NEW CONCEPT

```
spriteload spritenumber, filename
```

spriteload reads a picture file (BMP, PNG, GIF, JPG or JPEG) and turns it into a sprite. A new sprite has its center at 0, 0 and is hidden: put it in place with **spriteplace**, then show it with **spriteshow**. The transparent parts of a PNG picture stay see-through.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
spriteshow spritenumber
spritehide spritenumber
```

spriteshow makes a sprite visible on the graphics output. **spritehide** hides it: the sprite still exists, keeps its place, and can be shown again later.



NEW CONCEPT

```
spriteplace spritenumber, x, y
spritemove spritenumber, dx, dy
spriteplace spritenumber, x, y, scale, rotation
```

spriteplace puts the center of a sprite at *x*, *y*. **spritemove** moves it from where it is now: *dx* pixels to the right and *dy* pixels down. Negative numbers move it left and up. You may move a hidden sprite too; you see it when you show it.

A sprite can move off the screen: BASIC-256 doesn't stop it at the edge. If it must stay on the screen, your program has to check its position, as Program 89 does.

Like **imgload**, both statements may also take a *scale* and a *rotation*. With **spriteplace** they set the size and angle; with **spritemove** they are added to the ones it has.



NEW CONCEPT

```
spritex(spritenumber)   spritey(spritenumber)
spritew(spritenumber)   spriteh(spritenumber)
spritev(spritenumber)
```

These functions tell you about a sprite:

FUNCTION	RETURNS
`spritex(n)`	the x position of the center of the sprite
`spritey(n)`	the y position of the center of the sprite
`spritew(n)`	the width of the sprite in pixels
`spriteh(n)`	the height of the sprite in pixels
`spritev(n)`	true if the sprite is shown, false if it is hidden

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



TRY IT

Load a second ball in Program 89, *greenball.png*, as sprite 1, and let it bounce too, with its own *dx* and *dy*. Remember to change `spritedim 1`.

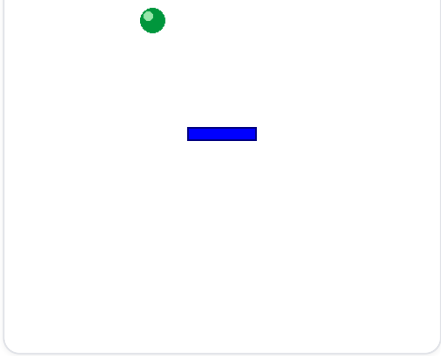
Program 90 has two sprites. Sprite 0, the bumper, stands still in the middle of the screen. Sprite 1, a green ball, bounces off the walls and off the bumper. Every time it hits the bumper, the program prints *bump*.

spritecollide tells when the two sprites touch. Then the ball changes its up-and-down direction. A ball moves several pixels at a time, so it may already be a little inside the bumper: the small **while** loop moves it straight up or down until it is free again. Without that loop the ball could stay stuck, turning round and round inside the bumper.

TEXT OUTPUT

```
bump
bump
bump
bump
```

GRAPHICS OUTPUT



spritebumper.kbs — BASIC-256

```
# spritebumper.kbs - two sprites that collide

clg white
spritedim 2

# the bumper stands still in the middle
spriteload 0, "paddle.png"
spriteplace 0, graphwidth / 2, graphheight / 2
spriteshow 0

# the ball moves
spriteload 1, "greenball.png"
spriteplace 1, 50, 50
spriteshow 1
dx = rand * 5 + 5
dy = rand * 5 + 5
r = spritew(1) / 2   # the radius of the ball

while true
   if spritex(1) <= r or spritex(1) >= graphwidth - r then
      dx = dx * -1
   end if
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
    if spritey(1) <= r or spritey(1) >= graphheight - r then
       dy = dy * -1
    end if
    if spritecollide(0, 1) then
       dy = dy * -1
       print "bump"
       # move the ball up or down, out of the bumper
       while spritecollide(0, 1)
          spritemove 1, 0, dy
       end while
    end if
    spritemove 1, dx, dy
    pause 0.02
end while
```

Program 90 · Two Sprites with Collision



NEW CONCEPT

```
spritecollide(spritenumber1, spritenumber2)
spritecollide(spritenumber1, spritenumber2, true)
```

spritecollide returns true when the two sprites touch or overlap, and false when they don't. A hidden sprite never collides. It treats each sprite as the rectangle around its picture, so round sprites, like the balls, can "touch" at their transparent corners. Add **true** as a third argument, and only pixels that are not transparent count: that is exact, but slower.

A sprite doesn't need a picture file: **spritepoly** makes one from a polygon, the same kind of list of points you used with **poly** and **stamp** in Chapter 9. Program 91 makes a blue arrow with a red outline that follows the mouse.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

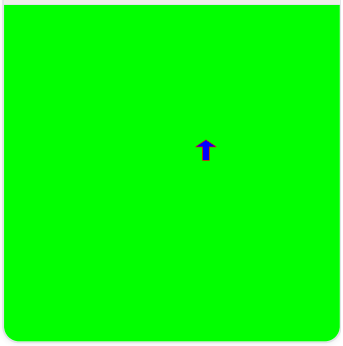
spritepoly.kbs BASIC-256

```
# spritepoly.kbs - a sprite made from a polygon
# that follows the mouse

spritedim 1
color red, blue
penwidth 1
spritepoly 0, {15, 0, 30, 10, 20, 10, 20, 30, 10, 30,
   10, 10, 0, 10}

clg green
while true
   if mousex > -1 then
      # the tip of the arrow is 15 pixels above its center
      spriteplace 0, mousex, mousey + 15
      spriteshow 0
   else
      spritehide 0
   end if
   pause 0.01
end while
```

GRAPHICS OUTPUT



The arrow, with the mouse pointer at its tip.

Program 91 · Creating a Sprite From a Polygon

Remember from Chapter 11 that **mousex** and **mousey** are −1 when the mouse is outside the graphics output. Then the program hides the arrow with **spritehide**; when the mouse comes back, **spriteshow** shows it again. The arrow is 30 pixels high and the sprite's position is its center, so the program places it 15 pixels lower than the mouse: then its tip is at the mouse pointer.



NEW CONCEPT

```
spritepoly spritenumber, {x1, y1, x2, y2, x3, y3 …}
```

spritepoly makes a sprite from a list of points: *x*, *y* pairs, as for **poly**. It is drawn with the color, fill color and **penwidth** you have chosen, and the sprite is made just big enough to hold the polygon and its outline. Like a loaded sprite, it starts hidden.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



SPRITES OR DRAWING?

Sprites float on their own layer above the graphics output. That is what makes them easy to move, but it also means that **clg** doesn't erase them, and **imgsave** doesn't save them: it saves only what was drawn. Draw the things that stay put, like a background, a score or a maze. Use sprites for the things that move.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

Paddle Ball with Sprites

Keep the ball in play with your paddle. Each time you hit it, it bounces back a little faster. When the ball reaches the bottom of the screen, the game is over and the program tells you how many times you hit it.

GRAPHICS OUTPUT



The paddle is sprite 0, made with **spritepoly**; the ball is sprite 1, the green ball from Program 85. Hold J to move the paddle left and K to move it right. The program uses **keypressed** from Chapter 12, so the paddle keeps moving as long as you hold the key, and stops at the edges of the screen.

The ball bounces off the sides and the top, each with its own sound. When it hits the paddle, both *dx* and *dy* are multiplied by 1.1, so it goes 10% faster, and *bounces* goes up by 1.

The game draws with **fastgraphics** (Chapter 8): the sprites move without being shown, and **refresh** shows the new picture once per round of the loop, so the ball and the paddle move smoothly together. Before each new sound, **soundstop** stops the one before, so the sounds don’t pile up when the ball bounces quickly.

sprite_paddleball.kbs — BASIC-256

```
# sprite_paddleball.kbs - a paddle ball game with sprites
# beep.mp3, click.mp3 and toggle.mp3 come with BASIC-256:
# Examples, Original_Examples, sprites folder
# (Mike Koenig, CC BY 3.0)

print "Paddle ball game"
print "Hold J or K down to move the paddle."
input "Press Enter to start > ", answer

clg white
fastgraphics
spritedim 2

# the paddle, made from a polygon
color blue, darkblue
penwidth 2
spritepoly 0, {0, 0, 80, 0, 80, 20, 70, 20, 70, 10,
   10, 10, 10, 20, 0, 20}
spriteplace 0, 250, 470
spriteshow 0

# the ball
spriteload 1, "greenball.png"
spriteplace 1, 250, 100
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
spriteshow 1

dx = rand * 2 + 2
dy = rand * 2 + 2
r = spritew(1) / 2   # the radius of the ball
p = spritew(0) / 2   # half the width of the paddle
bounces = 0

# play until the ball reaches the bottom
while spritey(1) < graphheight - r
   # move the paddle while J or K is held down
   if keypressed(asc("J")) and spritex(0) > p then
      spritemove 0, -6, 0
   end if
   if keypressed(asc("K")) and spritex(0) < graphwidth - p then
      spritemove 0, 6, 0
   end if
   if spritecollide(0, 1) then
      # bounce back and speed up
      dy = dy * -1.1
      dx = dx * 1.1
      bounces += 1
      soundstop
      soundplay "beep.mp3"
      # move the ball up, out of the paddle
      while spritecollide(0, 1)
         spritemove 1, 0, dy
      end while
   end if
   if spritex(1) <= r or spritex(1) >= graphwidth - r then
      dx = dx * -1
      soundstop
      soundplay "click.mp3"
   end if
   if spritey(1) <= r then
      dy = dy * -1
      soundstop
      soundplay "toggle.mp3"
   end if
   spritemove 1, dx, dy
   refresh
   pause 0.01
end while

print "You bounced the ball " + bounces + " times."
```

Program 92 · Big Program — Paddleball with Sprites

TEXT OUTPUT

```
Paddle ball game
Hold J or K down to move the paddle.
Press Enter to start > ⏎
You bounced the ball 10 times.
```

The text output. At ⏎ the player pressed Enter.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



TRY IT

Draw a score on the graphics output: after every hit, erase the top left corner with a white rectangle and write the number of *bounces* there with **text**. Because sprites float above the drawing, the ball still flies over it.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

e d i h e t i r p s v t b c o
j e v o m e t i r p s v c e i
y m k d o p e t u a n d s g n
l n j e j d d n f s t o p o s
o o n c e o i e s p i g r m p
p i w a y g l r o r a m i d r
e s h l e r l a u i w y t i i
t i e p v o o p n t d a e m t
i l r e a t c s d e n l l g e
r l u t s a e n s d u p o l s
p o t i g t t a t i o d a o h
s c c r m i i r o m s n d a o
m e i p i o r t p n o u v d w
i m p s n n p i l y i o o w s
h y t e u m s e l a c s r n h

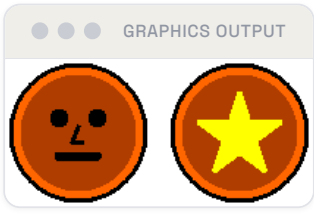
Find: collision, imgload, imgsave, picture, rotation, scale, soundplay, soundstop, soundwait, spritecollide, spritedim, spritehide, spriteload, spritemove, spriteplace, spritepoly, spriteshow, transparent



Problems

Try these — they only use what this chapter and the earlier ones covered.

1 Write a program that draws a coin with a face on it, on a graphics output of 100 × 100 pixels, and saves it as *head.png*. Then have the same program erase the screen, draw the back of the coin, and save it as *tail.png*. Make the coins your own design.

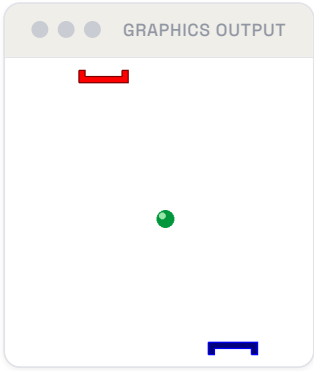


2 Now write a coin toss program that shows the result of a toss with the pictures from Problem 1. Pick a random number; if it is less than 0.5, show the heads picture, otherwise the tails picture.
For an extra challenge, keep showing random heads and tails until the user presses a key (use **key** from Chapter 12).

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

3 Record two sounds of your own: your voice saying “heads” and saying “tails”. Use a free program such as Audacity, or the voice recorder on your computer or phone, and save them as WAV or MP3 files in your program’s folder. Play them with **soundplay** in your program from Problem 2.

4 Change Program 92, Paddleball with Sprites, into a ping-pong game for two players. Add a third sprite: a paddle at the top for the second player, with two keys of its own to move it, for example A and S. The game ends when the ball gets past one of the paddles; then say who won.



Chapter 13 · Images, Sounds, and Sprites (formerly “Images, WAVs, and Sprites”). Modernised layout of the BASIC-256 beginner’s guide, updated for BASIC-256 2.3. Programs are numbered 84–92, continuing from Chapter 12. Corrected for 2.3: the obsolete `wavplay`, `wavwait` and `wavstop` are replaced by `soundplay`, `soundwait` and `soundstop`, which play in the background (asynchronously — the old text said “synchronous”); `imgsave` saves PNG, JPG and BMP files (GIF and `IMAGETYPE_GIF` don’t exist); a sprite is not stopped at the edge of the screen; `spritepoly` no longer needs the polygon to start at 0, 0; “showsprite” is `spriteshow`. New: Program 85, which makes the ball and paddle pictures, so the chapter needs no downloads apart from files that come with BASIC-256; transparency and the colour clear; where files are found and saved; what `clg` and `imgsave` do to sprites; the optional scale, rotation and exact-collision arguments. The dead links (photobucket, grsites.com, freesound.org) are gone: the Picasso photo is replaced by BitBot, and the sounds by recordings that come with BASIC-256 (Mike Koenig, soundbible.com, CC BY 3.0). Programs 89, 90 and 92 test the edges with the ball’s radius; Program 90 and the Big Program move the ball out of the paddle straight up or down so it can’t stick; Program 91 hides the arrow when the mouse leaves the output and puts its tip at the pointer. The Big Program uses `keypressed` and `fastgraphics` for smooth movement, keeps the paddle on the screen, ends when the ball reaches the bottom, speeds the ball up in both directions and uses `asc` and `+=` as in Chapter 12. All programs were run in BASIC-256 2.3; sprite pictures were drawn with `imgload` at the sprites’ places, because `imgsave` doesn’t save sprites. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Mascot: BitBot.