Vectors and PyGame

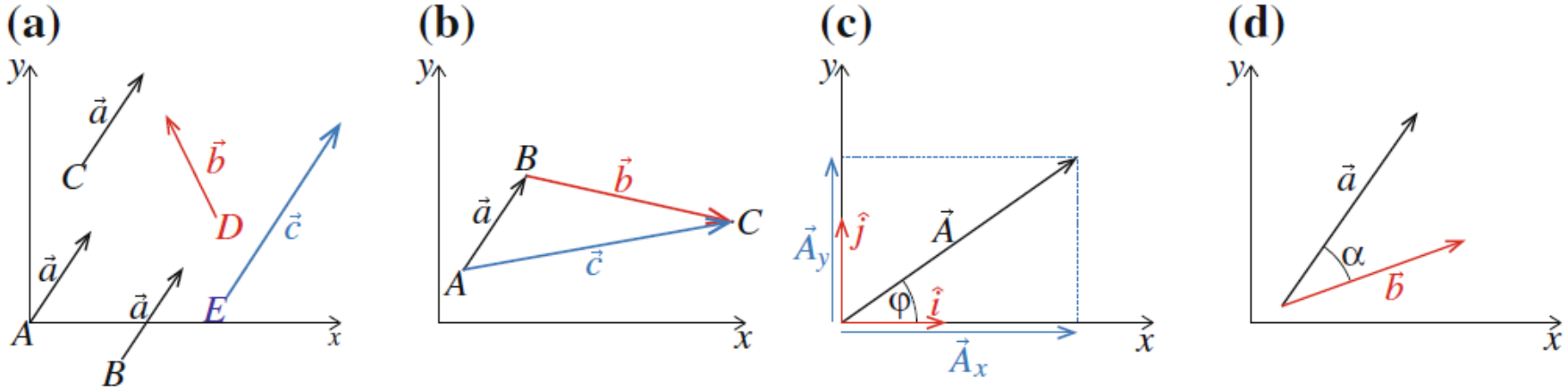
#3

Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

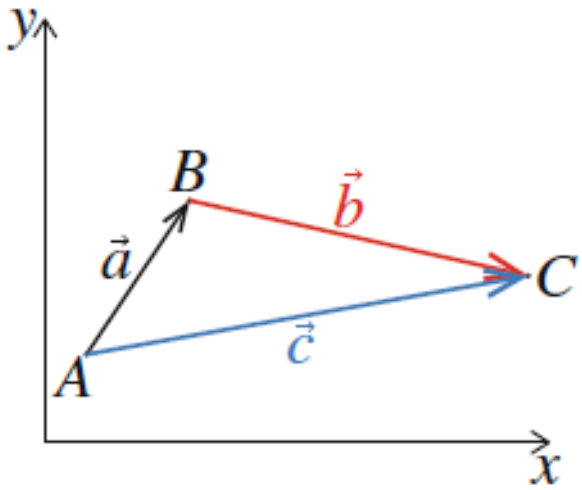
Vectors

(a) Illustration of vectors, (b) vector addition, (c) units vectors, decomposition and angle, (d) dot product



Vector Addition

Vector addition is intuitive for the addition of displacements: If you first move along the vector $\vec{a}$ from **A** to **B**, and then along the vector b from B to C, the net displacement is the vector:



$$\vec{c} = \vec{a} + \vec{b}$$

from point **A** to **C**. This geometric definition of vector addition is general, and we use it also for vectors that are not displacements: We find the sum of two vectors **a** and **b** geometrically by placing the tail of vector **b** at the tip of vector **a**. The sum is called the **resultant vector**.

From this definition, we realize that vector addition is **commutative**, the order of addition is arbitrary, and **associative**:

$$\vec{a} + \vec{b} = \vec{b} + \vec{a}$$

$$\vec{a} + (\vec{b} + \vec{c}) = (\vec{a} + \vec{b}) + \vec{c}$$

Scalar Multiplication

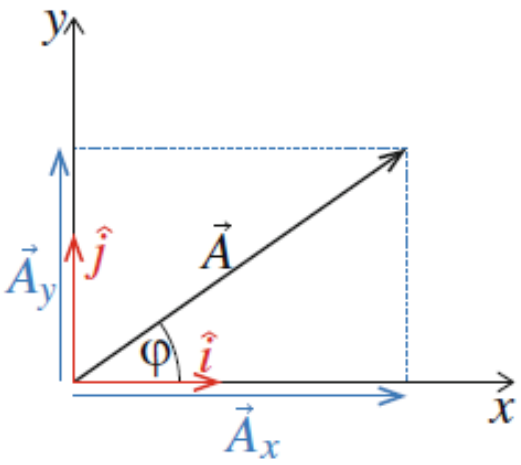
We can rescale the length of a vector by multiplying it with a scalar:

$$\vec{b} = 2\vec{a}$$

Vector **b** is twice as long as vector **a**, but still pointing in the same direction. If we multiply a vector by a **negative** number, we change the direction of the vector to point in the **opposite direction**.

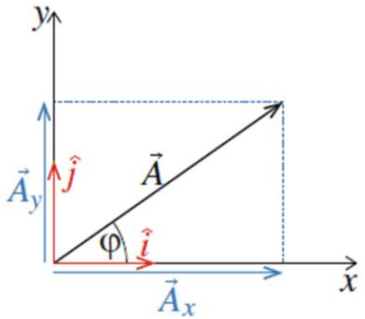
Vector Components

A coordinate system is a grid you choose to describe the world in numbers. Here, we use Cartesian coordinate systems, where the axes are orthogonal to each other. We describe the coordinate system by the position of the origin, O, and by unit vectors pointing along each axis: the x-, y-, and z-axis. The unit vectors are of unit length, of length 1, and do not have any unit. The unit vectors are orthogonal, they form 90◦ angles with each other. It is common to use the symbol ***i***, ***j***, and ***k*** for the ***unit vectors*** along the **x**, **y**, and **z**-axis respectively.



Any vector can be uniquely decomposed into a set of component vectors along each of the axes

$$\vec{A} = \overrightarrow{A_x} + \overrightarrow{A_y}$$



where each of the component vectors can be written in terms of the unit vector along the axis:

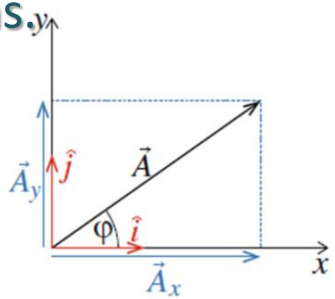
$$\overrightarrow{A_x} = A_x \boldsymbol{i},$$

$$\overrightarrow{A_y} = A_y \boldsymbol{j}$$

Here, the units of the vectors are in the **scalar** numbers ***Ax*** and ***Ay*** .

If you know the magnitude and direction of a vector, you can find the component of the vector from trigonometrical considerations.

$$\vec{A} = A_x \boldsymbol{i} + A_y \boldsymbol{j} = |\boldsymbol{A}| \boldsymbol{cos}\ \boldsymbol{\varphi i} + |\boldsymbol{A}| \boldsymbol{sin}\ \boldsymbol{\varphi j}$$



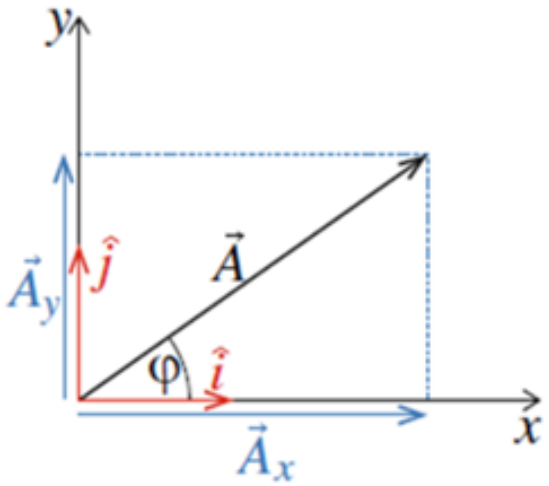
We may write a vector by using the unit vectors or by writing the vector directly in coordinate form:

$$\vec{A} = A_x \boldsymbol{i} + A_y \boldsymbol{j} + A_z \boldsymbol{k} = (A_x, A_y, A_z)$$

The Magnitude of a Vector Using Components

If the coordinate system is orthogonal, then all the axis are orthogonal to each other, and we can use Pythagoras' theorem to relate the magnitude to the vector to the components:

$$|\boldsymbol{A}| = \sqrt{A_x{}^2 + A_y{}^2}$$



Addition, Subtraction and Scalar Multiplication Using Components

A particular advantage of the component form is that addition, subtraction, and scalar multiplications can be done for each component independently.

$$\underbrace{A_x\boldsymbol{i} + A_y\boldsymbol{j}}_{\vec{\boldsymbol{A}}} + \underbrace{B_x\boldsymbol{i} + B_y\boldsymbol{j}}_{\vec{\boldsymbol{B}}} = \underbrace{(A_x + B_x)\boldsymbol{i} + (A_y + B_y)\boldsymbol{j}}_{= \vec{\boldsymbol{A}} + \vec{\boldsymbol{B}}}$$

$$\boldsymbol{c}\vec{\boldsymbol{A}} = \boldsymbol{c}(A_x\boldsymbol{i} + A_y\boldsymbol{j}) = \boldsymbol{c}A_y\boldsymbol{i} + \boldsymbol{c}B_y\boldsymbol{j}$$

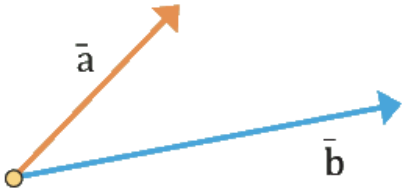
The Dot Product

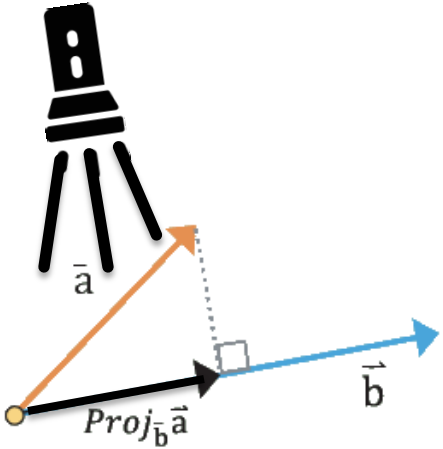
Often in game programming, you need to know how much one vector is in the direction of another vector. Mathematically, that requires finding the projection of a vector onto another vector. One example of the need for vector projections is to find the reflection of a vector off another vector. As you might imagine, this is commonly being done by the physics engine every time there is a collision to determine what new direction an object should move after the collision.

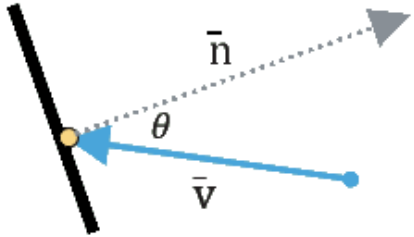
The Dot Product

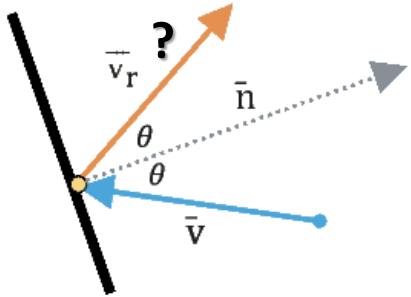
Often in game programming, you need to know how much one vector is in the direction of another vector. Mathematically, that requires finding the projection of a vector onto another vector. One example of the need for vector projections is to find the reflection of a vector off another vector. As you might imagine, this is commonly being done by the physics engine every time there is a collision to determine what new direction an object should move after the collision.

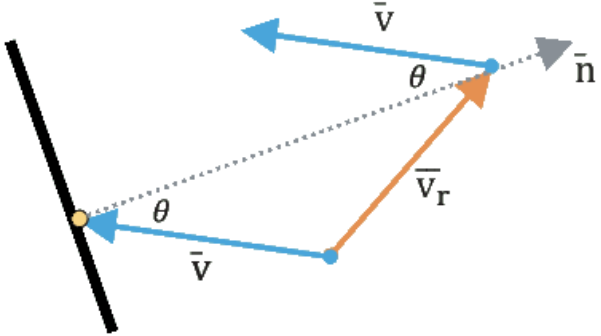
The Dot Product

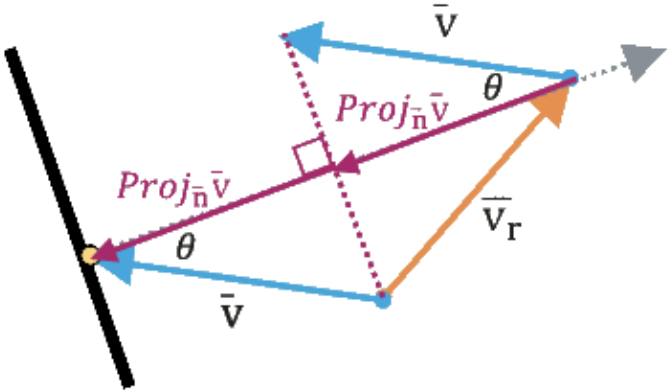


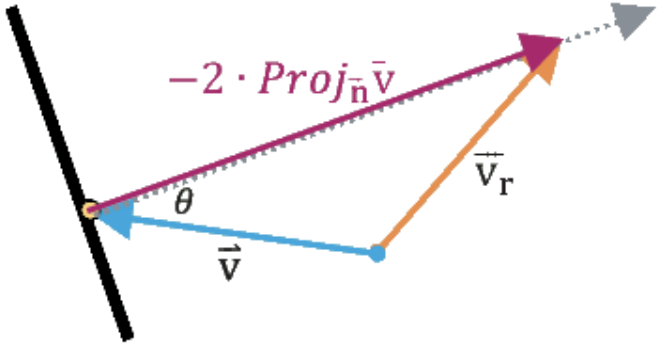










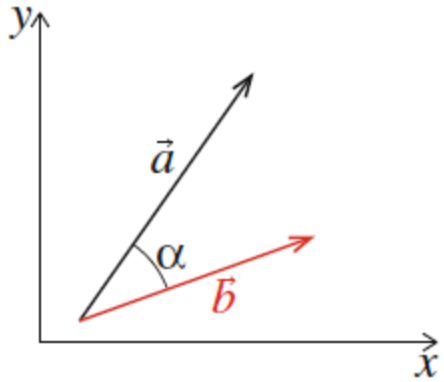


$$\overrightarrow{v_r} = \vec{v} - 2\,Proj_{\vec{n}}\vec{v}$$

The Dot Product

The dot product between two vectors A and B is defined as:

$$\vec{A}.\vec{B} = |A||B|cos\ \alpha$$



where α is the angle between the two vectors,
The dot product is linear:

$$(\vec{A} + \vec{B}).\vec{C} = \vec{A}.\vec{C} + \vec{B}.\vec{C}$$

And commutative:

$$\vec{A}.\vec{B} = \vec{B}.\vec{A}$$

The dot product depends on the angle α. When **two vectors are parallel** and point in the same direction, the dot product is **equal to the product of the magnitudes**. A particular useful property of the dot product is that the dot product of **two orthogonal vectors is zero**. As a result the dot product is simple on component form

$$\vec{A}.\vec{B} = (A_x\boldsymbol{i} + A_y\boldsymbol{j}).(B_x\boldsymbol{i} + B_y\boldsymbol{j})$$

$$\vec{A}.\vec{B} = A_xB_x\underbrace{\boldsymbol{i}.\boldsymbol{i}}_{=1} + A_xB_y\underbrace{\boldsymbol{i}.\boldsymbol{j}}_{=0} + A_yB_x\underbrace{\boldsymbol{j}.\boldsymbol{i}}_{=0} + A_yB_y\underbrace{\boldsymbol{j}.\boldsymbol{j}}_{=1}$$

$$\vec{A}.\vec{B} = A_xB_x + A_yB_y$$

What makes the dot product so useful, is that it can be used to decompose a vector onto a given set of unit vector—it can be used to find the components of a vector in any given coordinate system. A vector A can be written in component form as:

$$\vec{A} = A_x \boldsymbol{i} + A_y \boldsymbol{j} + A_z \boldsymbol{k} = (A_x, A_y, A_z)$$

How can we determine the components $\boldsymbol{A_x}$, $\boldsymbol{A_y}$, and $\boldsymbol{A_z}$? We find them by using to dot product, and remembering that the unit vectors are orthogonal.

$$\vec{A}.\boldsymbol{i} = A_x \underbrace{\boldsymbol{i}.\boldsymbol{i}}_{=1} + A_y \underbrace{\boldsymbol{j}.\boldsymbol{i}}_{=0} + A_z \underbrace{\boldsymbol{k}.\boldsymbol{i}}_{=0} = A_x$$

The component of a vector $\boldsymbol{A}$ can be found by dot-multiplication with the unit vectors $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$:

$$A_x = \vec{A}.\boldsymbol{i} \qquad A_y = \vec{A}.\boldsymbol{j} \qquad A_z = \vec{A}.\boldsymbol{k}$$

Pygame Library

To make graphics easier to work with, we'll use "**Pygame**"

- Draw graphic shapes
- Display bitmapped images
- Animate
- Interact with the keyboard, mouse, and gamepad
- Play sound
- Detect when objects collide

```
# Import a library of functions called 'pygame'
import pygame
# Initialize the game engine
pygame.init()
```

Colors

Colors are defined in a list of three colors: red, green, and blue.
Each element of the RGB triad is a number ranging from 0 to 255. Zero means there is none of the color, and 255 tells the monitor to display as much of the color as possible. The colors combine in an additive way, so if all three colors are specified, the color on the monitor appears white.

```
# Define some colors
BLACK = ( 0, 0, 0)
WHITE = ( 255, 255, 255)
GREEN = ( 0, 255, 0)
RED = ( 255, 0, 0)
```

Open a Window

So far, the programs we have created only printed text out to the screen or plot some graphics. The code to open a window is not complex. Below is the required code, which creates a window sized to a width of **700 pixels and a height of 500:**

```
size = (700, 500)
screen = pygame.display.set_mode(size)
```

To set the title of the window (which is shown in the title bar), use the following line of code:

```
pygame.display.set_caption("BCO 611 Physics in CAG")
```

Interacting with the User

```
# Loop until the user clicks the close button.
done = False
# Used to manage how fast the screen updates
clock = pygame.time.Clock()
# -------- Main Program Loop -----------
while not done:
    # --- Main event loop
    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Exit this loop
    # --- Game logic should go here
    # --- Drawing code should go here
    # First, clear the screen to white. Don't put other drawing commands
    # above this, or they will be erased with this command.
    screen.fill(WHITE)
    # --- Go ahead and update the screen with what we've drawn.
    pygame.display.flip()
    # --- Limit to 60 frames per second
    clock.tick(60)
```

Main Program Loop

The Event Processing Loop

The for loop processes all the events in a list.

```
for event in pygame.event.get(): # get events from the queue
    if event.type == pygame.QUIT:
        print("User asked to quit.")
    elif event.type == pygame.KEYDOWN:
        print("User pressed a key.")
    elif event.type == pygame.KEYUP:
        print("User let go of a key.")
    elif event.type == pygame.MOUSEBUTTONDOWN:
        print("User pressed a mouse button")
```

The Event Processing Loop

QUIT	none
ACTIVEEVENT	gain, state
KEYDOWN	key, mod, unicode, scancode
KEYUP	key, mod, unicode, scancode
MOUSEMOTION	pos, rel, buttons, touch
MOUSEBUTTONUP	pos, button, touch
MOUSEBUTTONDOWN	pos, button, touch
JOYAXISMOTION	instance_id, axis, value
JOYBALLMOTION	instance_id, ball, rel
JOYHATMOTION	instance_id, hat, value
JOYBUTTONUP	instance_id, button
JOYBUTTONDOWN	instance_id, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	none
USEREVENT	code

The Event Processing Loop

Since pygame 2.0.1, there are a new set of events, called window events.

Event type	Short description
WINDOWSHOWN	Window became shown
WINDOWHIDDEN	Window became hidden
WINDOWEXPOSED	Window got updated by some external event
WINDOWMOVED	Window got moved
WINDOWRESIZED	Window got resized
WINDOWSIZECHANGED	Window changed it's size
WINDOWMINIMIZED	Window was minimized
WINDOWMAXIMIZED	Window was maximized
WINDOWRESTORED	Window was restored
WINDOWENTER	Mouse entered the window
WINDOWLEAVE	Mouse left the window
WINDOWFOCUSGAINED	Window gained focus
WINDOWFOCUSLOST	Window lost focus
WINDOWCLOSE	Window was closed
WINDOWTAKEFOCUS	Window was offered focus
WINDOWHITTEST	Window has a special hit test

Processing Each Frame

The basic logic and order for each frame of the game:

- **While not done:**
  - **For each event (keypress, mouse click, etc.):**
    - **Use a chain of if statements to run code to handle each event.**
- **Run calculations to determine where objects move, what happens when objects collide, etc.**
- **Clear the screen.**
- **Draw everything.**

It makes the program easier to read and understand if these steps aren't mixed together. Don't do some calculations, some drawing, some more calculations, some more drawing.

Ending the Program

Clicking the “close” button of a window while running this Pygame program in IDLE will still cause the program to crash. This is a hassle because it requires a lot of clicking to close a crashed program. The problem is, even though the loop has exited, the program hasn't told the computer to close the window. By calling the command below, the program will close any open windows and exit as desired.

```
pygame.quit()
```

Clearing the Screen

The following code clears whatever might be in the window with a white background. Remember that the variable WHITE was defined earlier as a list of three RGB values.

```
# Clear the screen and set the screen background
screen.fill(WHITE)
```

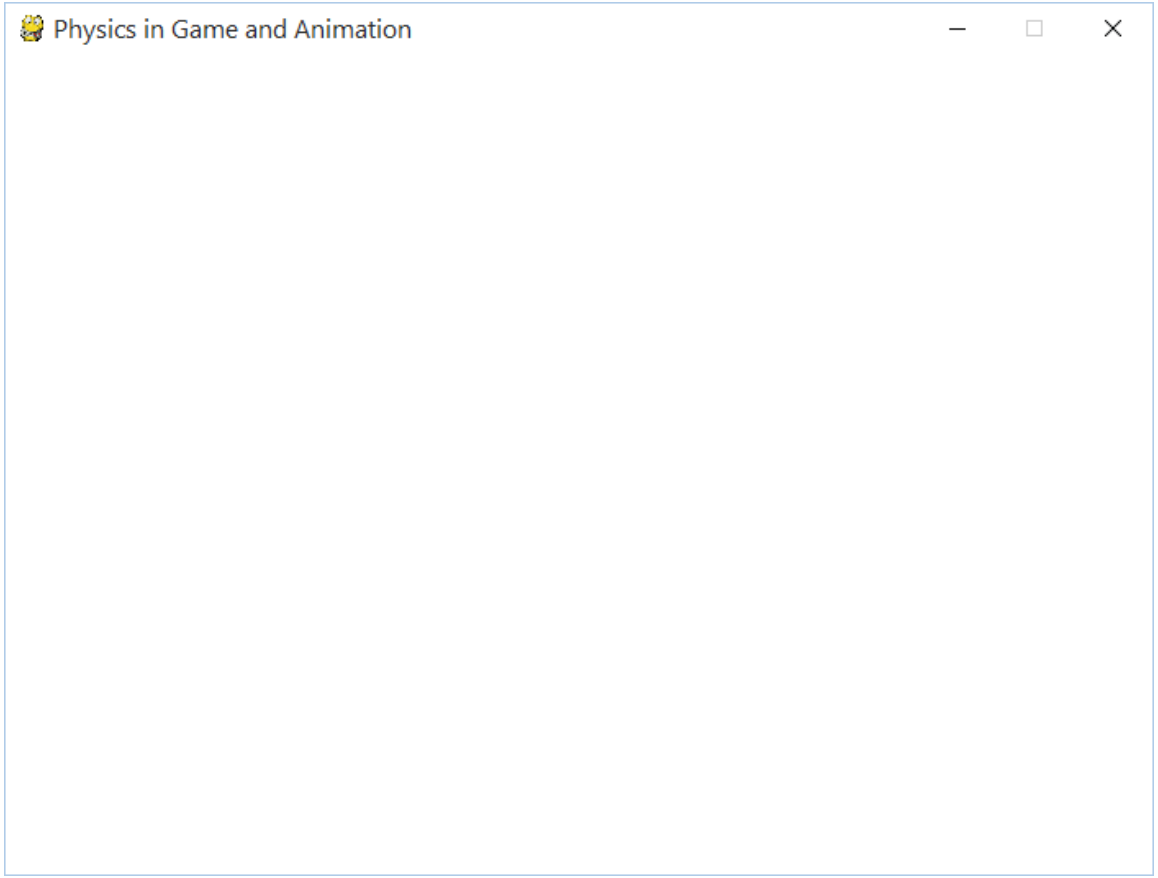
Open a Blank Window

```
import pygame
# Define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
pygame.init()
# Set the width and height of the screen [width, height]
size = (700, 500)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Physics in Game and Animation")
# Loop until the user clicks the close button.
done = False
# Used to manage how fast the screen updates
clock = pygame.time.Clock()
```

Open a Blank Window

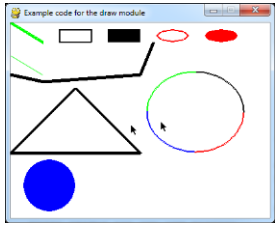
```
# -------- Main Program Loop -----------
while not done:
    # --- Main event loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    # --- Game logic should go here
    # --- Drawing code should go here
    # First, clear the screen to white. Don't put other drawing commands
    # above this, or they will be erased with this command.
    screen.fill(WHITE)
    # --- Go ahead and update the screen with what we've drawn.
    pygame.display.flip()
    # --- Limit to 60 frames per second
    clock.tick(60)
# Close the window and quit.
# If you forget this line, the program will 'hang'
# on exit if running from IDLE.
pygame.quit()
```

PHYSICS in COMPUTER ANIMATIONS and GAMES



Drawing

Here is a list of things that you can draw: `http://www.pygame.org/docs/ref/draw.html`

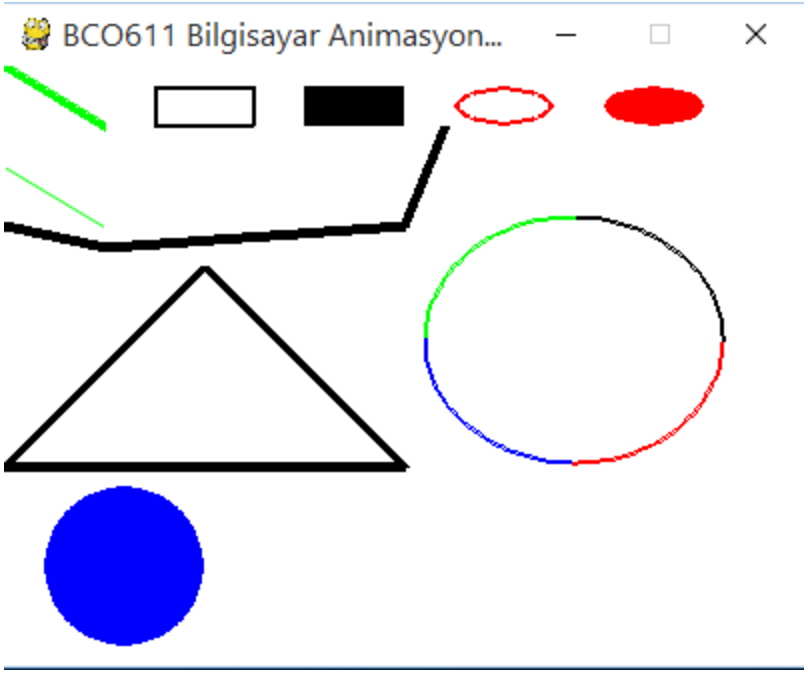


`pygame module for drawing shapes`

`pygame.draw.rect`	— draw a rectangle shape
`pygame.draw.polygon`	— draw a shape with any number of sides
`pygame.draw.circle`	— draw a circle around a point
`pygame.draw.ellipse`	— draw a round shape inside a rectangle
`pygame.draw.arc`	— draw a partial section of an ellipse
`pygame.draw.line`	— draw a straight line segment
`pygame.draw.lines`	— draw multiple contiguous line segments
`pygame.draw.aaline`	— draw fine antialiased lines
`pygame.draw.aalines`	— draw a connected sequence of antialiased lines

Drawing Lines

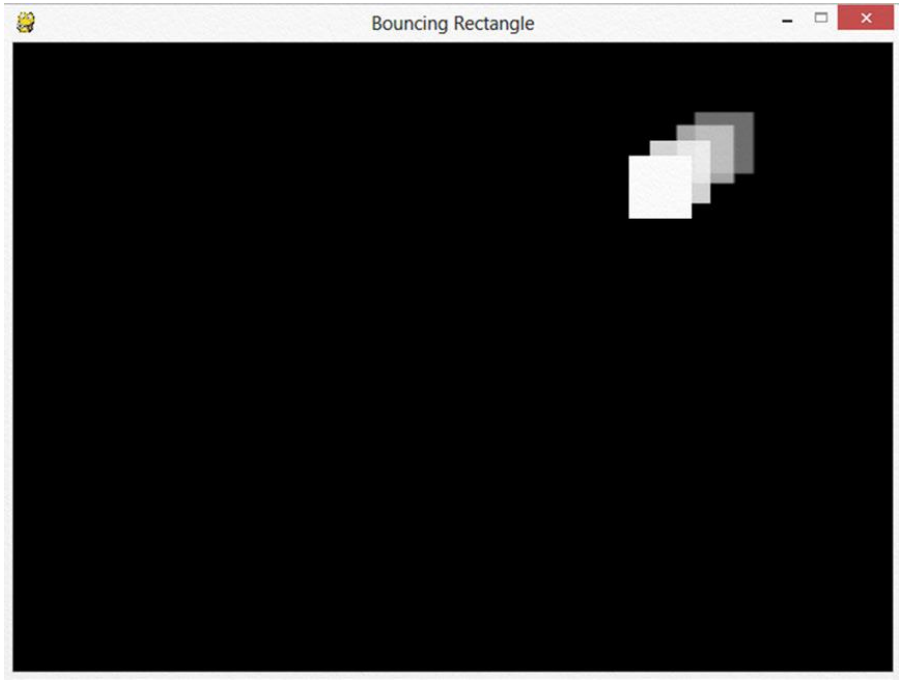
```
# Draw on the screen a green line from (0, 0) to (100, 100)
# that is 5 pixels wide.
pygame.draw.line(screen, GREEN, [0, 0], [100, 100], 5)
# Draw on the screen several lines from (0, 10) to (100, 110)
# 5 pixels wide using a while loop
y_offset = 0
while y_offset < 100:
    pygame.draw.line(screen,RED,[0,10+y_offset],[100,110+y_offset],5)
    y_offset = y_offset + 10
# Draw on the screen several lines from (0,10) to (100,110)
# 5 pixels wide using a for loop
for y_offset in range(0, 100, 10):
    pygame.draw.line(screen,RED,[0,10+y_offset],[100,110+y_offset],5)
```



`draw_pygame_example.py`

Animation

We will put together a program to bounce a white rectangle around a screen with a black background.



Animation

First step: start with the base template and flip the background color from white to black.

```
screen.fill(BLACK)
```

Next up, draw the rectangle we plan to animate. A simple rectangle will suffice. This code should be placed after clearing the screen, and before flipping it.

```
pygame.draw.rect(screen, WHITE, [50, 50, 50, 50])
```

To move the rectangle to the right, x can be increased by one each frame. This code is close, but it does not quite do it:

```
rect_x = 50
pygame.draw.rect(screen, WHITE, [rect_x, 50, 50, 50])
rect_x += 1
```

Animation

The problem with the code is that **rect_x** is reset back to 50 each time through the loop. To fix this problem, move the initialization of **rect_x** up outside of the loop. This next section of code will successfully slide the rectangle to the right

```
# Starting x position of the rectangle
# Note how this is outside the main while loop.
rect_x = 50
# -------- Main Program Loop -----------
while not done:
    for event in pygame.event.get():  # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Flag that we are done so we exit this loop
    # Set the screen background
    screen.fill(BLACK)
    pygame.draw.rect(screen, WHITE, [rect_x, 50, 50, 50])
    rect_x += 1
```

Animation

We can expand this code and increase both x and y, causing the square to move the box faster both down and right.

```
# Starting x position of the rectangle
rect_x = 50
rect_y = 50
# -------- Main Program Loop -----------
while not done:
    for event in pygame.event.get():  # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Flag that we are done so we exit this loop
    # Set the screen background
    screen.fill(BLACK)
    pygame.draw.rect(screen, WHITE, [rect_x, 5 rect_y, 50, 50])
    rect_x += 5
    rect_y += 5
```

Animation

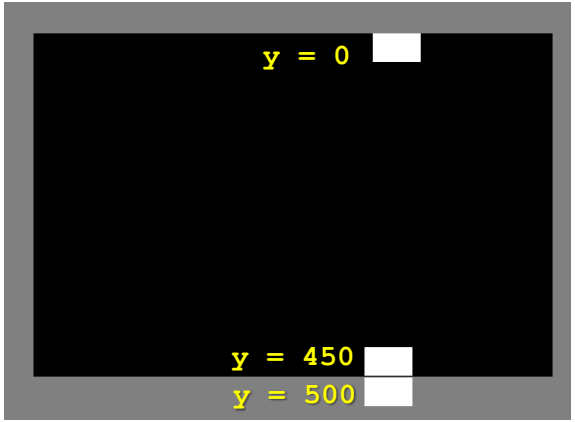
The direction and speed of the box's movement can be stored in a vector.

```
# Starting x and y position of the rectangle
rect_x = 50
rect_y = 50
# Speed and direction of rectangle
rect_change_x = 5
rect_change_y = 5
# -------- Main Program Loop -----------
while not done:
    for event in pygame.event.get():  # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Flag that we are done so we exit this loop
    # Set the screen background
    screen.fill(BLACK)
    pygame.draw.rect(screen, WHITE, [rect_x, 5 rect_y, 50, 50])
    rect_x += rect_change_x
    rect_y += rect_change_y
```

Animation

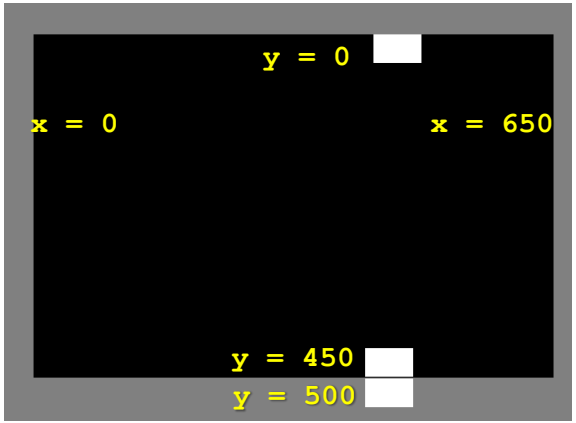
Once the box hits the edge of the screen it will keep going. Nothing makes the rectangle bounce off the edge of the screen. To reverse the direction so the rectangle travels towards the right, **rect_change_y** needs to change from 5 to -5 once the rectangle gets to the bottom side of the screen. The rectangle is at the bottom when **rect_y** is greater than the height of the screen. The code below can do the check and reverse the direction:

```
# Bounce the rectangle if needed
if rect_y > 450:
    rect_change_y = rect_change_y * -1
```

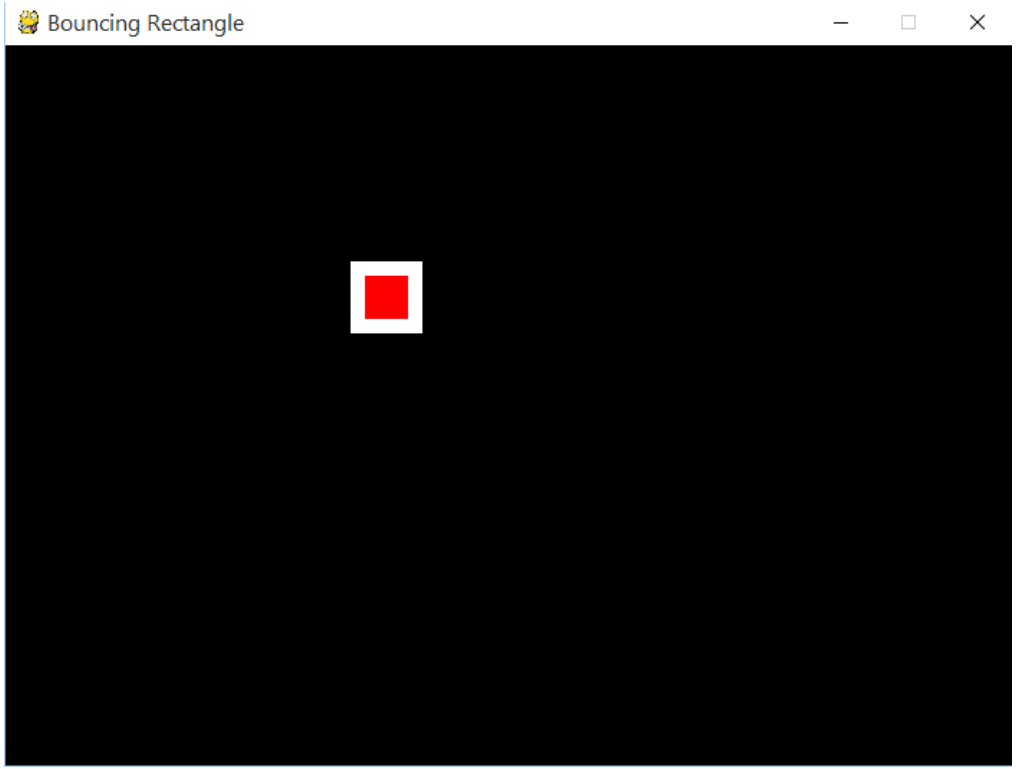


Animation

```
# Bounce the rectangle if needed
if rect_y > 450 or rect_y < 0:
    rect_change_y = rect_change_y * -1
if rect_x > 650 or rect_x < 0:
    rect_change_x = rect_change_x * -1
```




```
# Draw a red rectangle inside the white one
pygame.draw.rect(screen, WHITE, [rect_x, rect_y, 50, 50])
pygame.draw.rect(screen, RED, [rect_x + 10, rect_y + 10 ,30, 30])
```

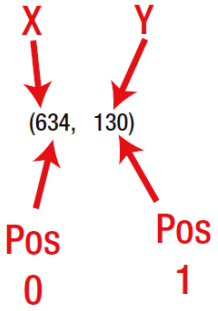


bouncing_rectangle.py

Mouse

It takes one line of code to get the mouse coordinates.

```
pos = pygame.mouse.get_pos()
```




```
# Game logic
pos = pygame.mouse.get_pos()
x = pos[0]
y = pos[1]
```

The mouse can be hidden by using the following code right before the main program loop

```
# Hide the mouse cursor
pygame.mouse.set_visible(False)
```

Keyboard

Controlling with the keyboard is a bit more complex. Inside the main while loop of the program, we need to add some items to our event processing loop. In addition to looking for a **`pygame.QUIT`** event, the program needs to look for keyboard events. An event is generated each time the user presses a key. To start with, set the location and speed before the main loop starts:

```
# Speed in pixels per frame
x_speed = 0
y_speed = 0
# Current position
x_coord = 10
y_coord = 10
```

Keyboard

A **pygame.KEYDOWN** event is generated when a key is pressed down. A **pygame.KEYUP** event is generated when the user lets up on a key.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT: done = True
    # User pressed down on a key
    elif event.type == pygame.KEYDOWN:
    # Figure out if it was an arrow key. If so adjust speed.
        if event.key == pygame.K_LEFT: x_speed = -3
        elif event.key == pygame.K_RIGHT: x_speed = 3
        elif event.key == pygame.K_UP: y_speed = -3
        elif event.key == pygame.K_DOWN: y_speed = 3
    # User let up on a key
    elif event.type == pygame.KEYUP:
    # If it is an arrow key, reset vector back to zero
        if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT: x_speed = 0
        elif event.key == pygame.K_UP or event.key == pygame.K_DOWN: y_speed = 0
# Move the object according to the speed vector.
x_coord += x_speed
y_coord += y_speed
```

Keyboard

https://www.pygame.org/docs/ref/key.html

Pygame Code	ASCII	Common Name
K_BACKSPACE	\b	backspace
K_RETURN	\r	return
K_TAB	\t	tab
K_ESCAPE ^[	escape	
K_SPACE	space	
K_COMMA	,	comma sign
K_MINUS	-	minus
K_PERIOD	.	period slash
K_SLASH	/	forward
K_0	0	0
K_1	1	1
…	…	…
K_9	9	9
K_SEMICOLON	;	semicolon sign
K_EQUALS	=	equals sign

Keyboard

https://www.pygame.org/docs/ref/key.html

Pygame Code	ASCII	Common Name
K_LEFTBRACKET	[	left
K_RIGHTBRACKET	]	right
K_BACKSLASH	\	backslash bracket
K_BACKQUOTE	`	grave
K_a	a	a
K_b	b	b
…	…	…
K_y	y	y
K_z	z	z
K_DELETE		delete
K_KP0	keypad	0
K_KP1	keypad	1
…	…	…
K_KP8	keypad	8
K_KP9	keypad	9

Keyboard

https://www.pygame.org/docs/ref/key.html

Pygame Code	ASCII	Common Name
K_UP	up	arrow
K_DOWN	down	arrow
K_RIGHT	right	arrow
K_LEFT	left	arrow
K_RSHIFT	right	shift
K_LSHIFT	left	shift
K_RCTRL	right	ctrl
K_LCTRL	left	ctrl
K_RALT	right	alt
K_LALT	left	alt

Keyboard

https://www.pygame.org/docs/ref/key.html

Pygame Code	ASCII	Common Name
K_LEFTBRACKET	[	left
K_RIGHTBRACKET	]	right
K_BACKSLASH	\	backslash bracket
K_BACKQUOTE	`	grave
K_a	a	a
K_b	b	b
…	…	…
K_y	y	y
K_z	z	z
K_DELETE		delete
K_KP0	keypad	0
K_KP1	keypad	1
…	…	…
K_KP8	keypad	8
K_KP9	keypad	9

Game Controller

Game controllers require a different set of code, but the idea is still simple. To begin, check to see if the computer has a joystick, and initialize it before use.

```
# Current position
x_coord = 10
y_coord = 10
# Count the joysticks the computer has
joystick_count = pygame.joystick.get_count()
if joystick_count == 0:
    # No joysticks!
    print("Error, I didn't find any joysticks.")
else:
    # Use joystick #0 and initialize it
    my_joystick = pygame.joystick.Joystick(0)
    my_joystick.init()
```

Game Controller

A joystick will return two floating-point values. If the joystick is perfectly centered it will return (0, 0).

Game Controller

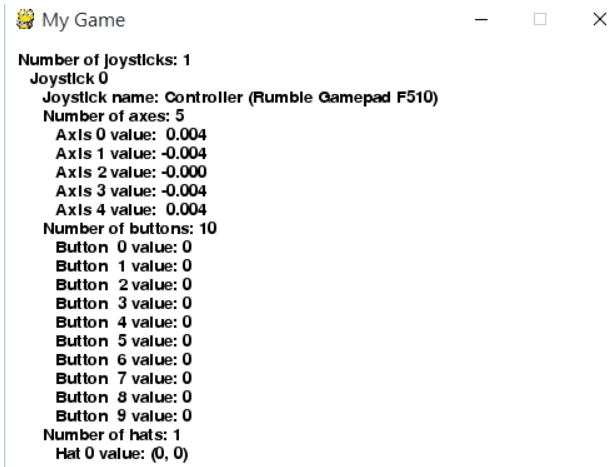
Game Controller

The values of the joystick returns may be multiplied according to how far an object should move.

```
# This goes in the main program loop!
# As long as there is a joystick
if joystick_count != 0:
    # This gets the position of the axis on the game controller
    # It returns a number between -1.0 and +1.0
    horiz_axis_pos = my_joystick.get_axis(0)
    vert_axis_pos = my_joystick.get_axis(1)
    # Move x according to the axis.
    # We multiply by 10 to speed up the movement.
    # Convert to an integer because we can't draw at pixel 3.5, just 3 or 4.
    x_coord = x_coord + int(horiz_axis_pos * 10)
    y_coord = y_coord + int(vert_axis_pos * 10)
# Clear the screen
screen.fill(BLACK)
# Draw the item at the proper coordinates
```

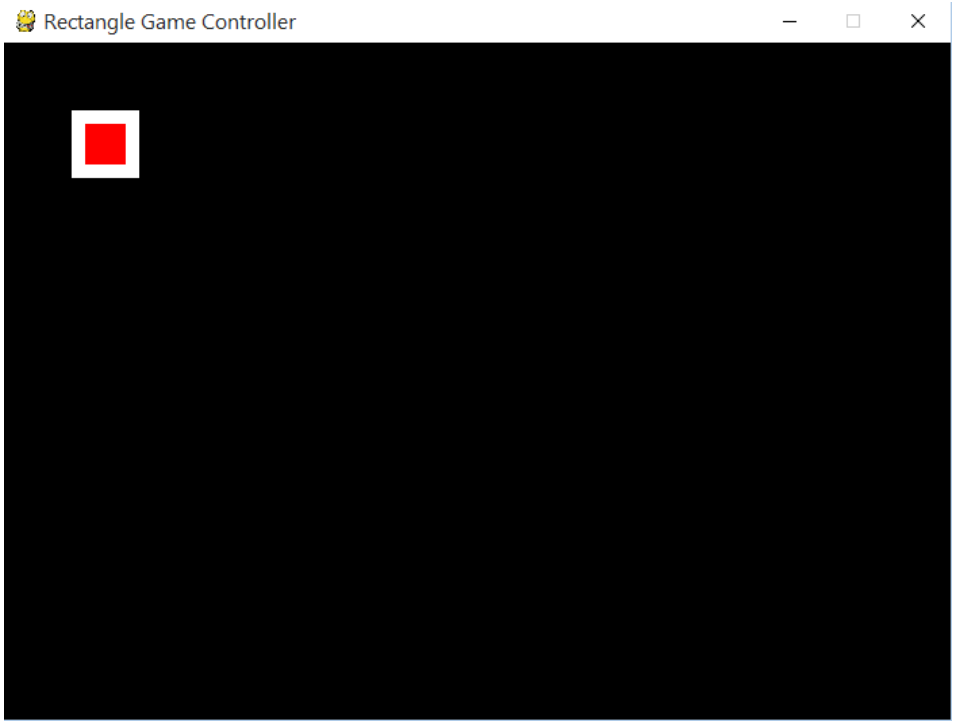
Game Controller

Controllers have a lot of joysticks, buttons, and even hat switches. Below is an example program and screenshot that prints everything to the screen showing what each game controller is doing. Take heed that game controllers must be plugged in before this program starts, or the program can't detect them.



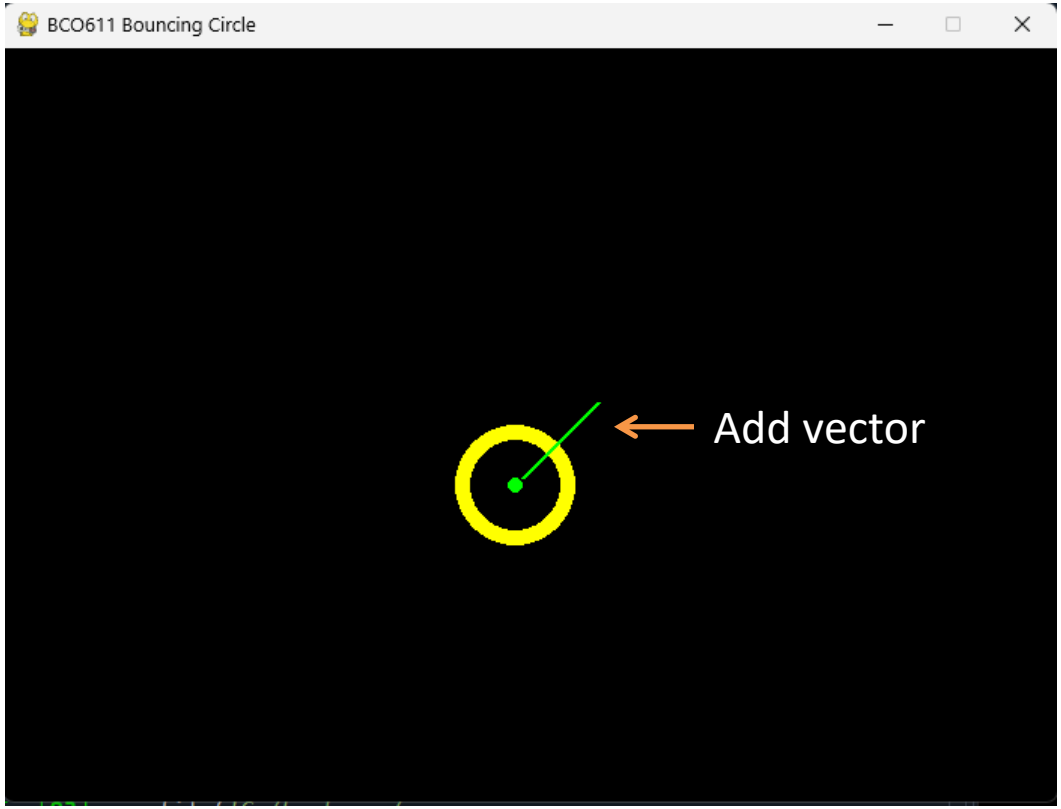
joystick_calls.py

Game Controller



`move_rectangle_game_controller.py`

Homework



`bouncing_sphere.py`

Sounds in Pygame

In this section we'll play a sound when the mouse button is clicked. Like images, sounds must be loaded before they are used. This should be done once sometime before the main program loop. The following command loads a sound file and creates a variable named `click_sound` to reference it

```
click_sound = pygame.mixer.Sound("sound.ogg")
```

We can play the sound by using the following command:

```
click_sound.play()
```

Sounds in Pygame

But where do we put this command? If we put it in the main program loop it will play it 20 times or so per second. Really annoying. We need a trigger. Some action occurs, then we play the sound. For example, this sound can be played when the user hits the mouse button with the following code:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        done = True
    elif event.type == pygame.MOUSEBUTTONDOWN:
        click_sound.play()
```

Sounds in Pygame

Uncompressed sound files usually end in .wav. These files are larger than other formats because no algorithm has been run on them to make them smaller. There is also the ever popular `.mp3` format, although that format has patents that can make it undesirable for certain applications. Another format that is free to use is the **OGG Vorbis** format that ends in `.ogg`. Pygame does not play all `.wav` files that can be found on the Internet. If you have a file that isn't working, you can try using the program `http://sourceforge.net/projects/audacity` to convert it to an **ogg-vorbis** type of sound file that ends in .ogg. Great places to find free sounds to use in your program.
OpenGameArt.org and **www.freesound.org**

Setting a Background Image in Pygame

Any bitmap images used in a game should already be sized for how it should appear on the screen. Don't take a 5000x5000 pixel image from a high-resolution camera and then try to load it into a window only 800x600. Use a graphics program (even MS Paint will work) and resize/crop the image before using it in your Python program. Loading an image is a simple process and involves only one line of code.

```
pygame.image.load("background.jpg")
```

This file must be located in the same directory that the Python program is in, or the computer will not find it.

Setting a Background Image in Pygame

That code may load the image, but we have no way to reference that image and display it! We need a variable set equal to what the load() command returns.

```
background_image = pygame.image.load("background.jpg")
```

Finally, the image needs to be converted to a format that pygame can more easily work with. To do that, we append `.convert()` to the command to call the convert function.

```
background_image = pygame.image.load("background.jpg").convert()
```

Setting a Background Image in Pygame

Loading the image should be done before the main program loop. While it would be possible to load it in the main program loop, this would cause the program to fetch the image from the disk 20 or so times per second. This is completely unnecessary. It is only necessary to do it once at program startup. To display the image use the `blit` command. This blits the image bits to the screen. The blit command is a method in the screen variable, so we need to start our command by `screen.blit`. Next, we need to pass the image to blit and where to blit it. This command should be done inside the loop so the image gets drawn each frame.

```
screen.blit(background_image, [0, 0])
```

Moving an Image in Pygame

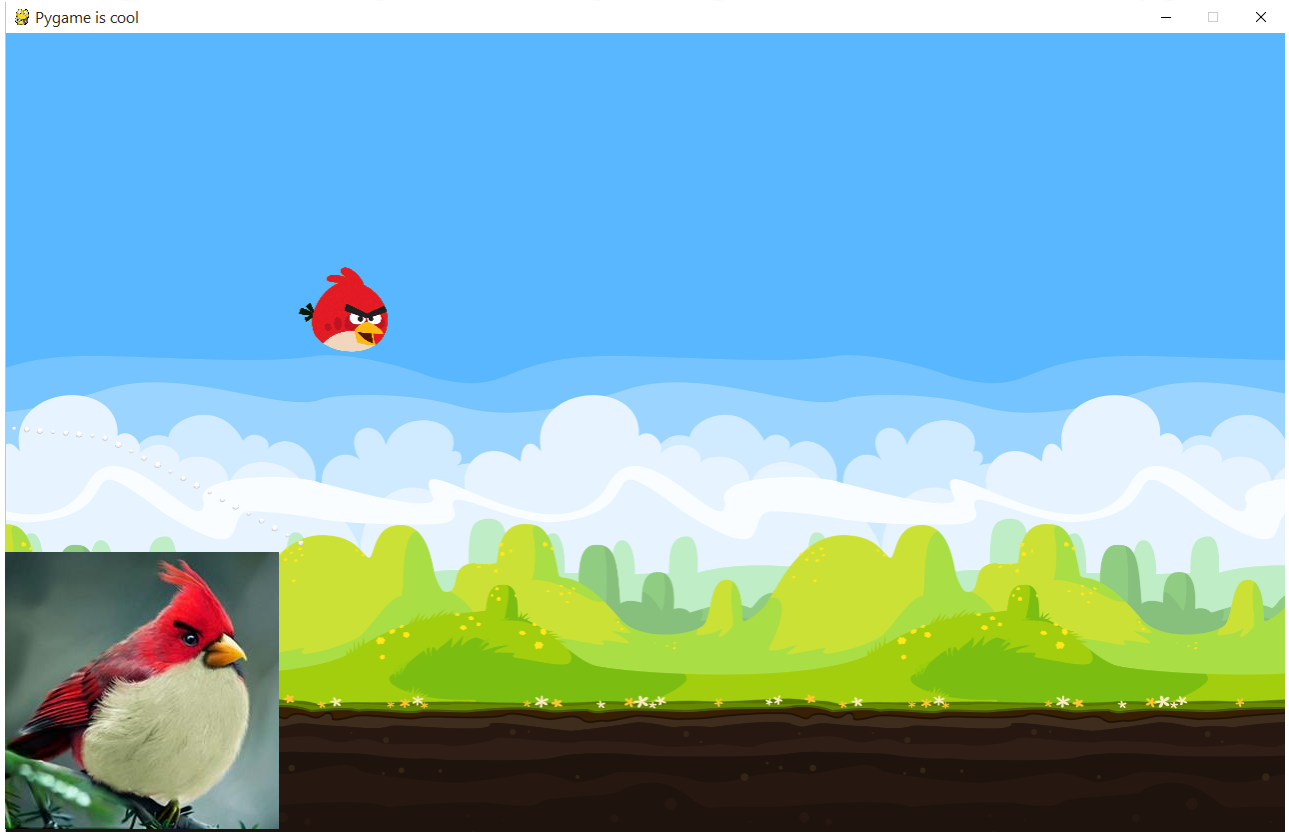
Now we want to load an image and move it around the screen. The image for the bird can be downloaded from the course web site, or you can find a `.gif` or `.png` that you like with a white or black background. Don't use a `.jpg`.

```
player_image = pygame.image.load("background.jpg").convert()
```

Finally, the image needs to be converted to a format that pygame can more easily work with. To do that, we append `.convert()` to the command to call the convert function.

```
background_image = pygame.image.load("background.jpg").convert()
```

Background Image, moving Image and Sounds in Pygame



PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Call this function so the Pygame library can initialize itself
pygame.init()
# Create an 1280x800 sized screen
screen = pygame.display.set_mode([1280, 800])
# This sets the name of the window
pygame.display.set_caption("BCO611 is cool")
clock = pygame.time.Clock()
# Before the loop, load the sounds
click_sound = pygame.mixer.Sound("voyyy.ogg")
# Set positions of graphics
background_position = [0, 0]
# Load and set up graphics
background_image = pygame.image.load("background.png").convert()
player_image = pygame.image.load("red_sprite_small.png").convert()

done = False
```

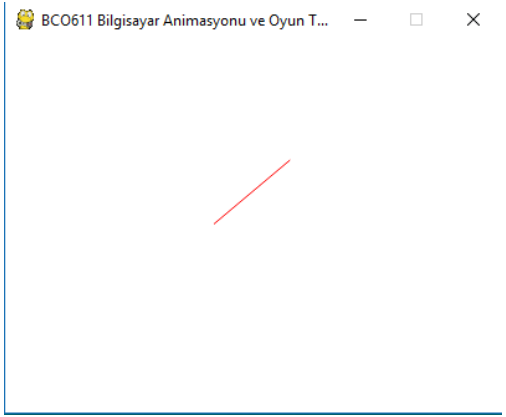
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        elif event.type == pygame.MOUSEBUTTONDOWN:
            click_sound.play()
    # Copy image to screen:
    screen.blit(background_image, background_position)
    # Get the current mouse position. This returns the position
    # as a list of two numbers
    player_position = pygame.mouse.get_pos()
    x = player_position[0] - 45
    y = player_position[1] - 45
    # Copy image to screen
    screen.blit(player_image, [x, y])
    pygame.display.flip()
    clock.tick(20)
pygame.quit ()
```

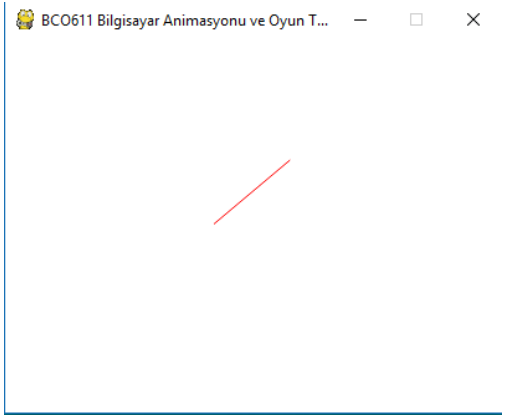
Limit the Red's movement

Vectors in Pygame

```
import pygame
import math

# Initialize the game engine
pygame.init()
# Define the colors we will use in RGB format
black = (  0,   0,    0)
white = (255, 255, 255)
blue =  (  0,   0, 255)
green = (  0, 255,   0)
red =   (255,   0,   0)
# Set the height and width of the screen
size = [400, 300]
screen = pygame.display.set_mode(size)
pygame.display.set_caption("BCO611 Bilgisayar Animasyonu ve Oyun Teknolojilerinde Fizik")
#Loop until the user clicks the close button.
done = False
clock = pygame.time.Clock()
angle = 0
mag = 10
# control how held keys are repeated
pygame.key.set_repeat(1, 50)
x, y = pygame.mouse.get_pos()
```



```
while not done:

    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Flag that we are done so we exit this loop
        if event.type == pygame.KEYDOWN:
            # Figure out if it was an arrow key. If so adjust speed.
            if event.key == pygame.K_DOWN: angle+= 10
            elif event.key == pygame.K_UP: angle-= 10
            elif event.key == pygame.K_RIGHT: mag+= 5
            elif event.key == pygame.K_LEFT: mag-= 5
        elif event.type == pygame.MOUSEBUTTONDOWN:
            print("User pressed a mouse button")
            x, y = pygame.mouse.get_pos() # get the initial coordinates

    # Clear the screen and set the screen background
    screen.fill(white)
    # Draw on the screen a  line from (x1,y1) to (x2, y2)
    pygame.draw.aaline(screen, red, [x, y], [x+mag*math.cos(math.radians(angle)),\
                                             y+mag*math.sin(math.radians(angle))], True)
    # This MUST happen after all the other drawing commands
    pygame.display.flip()
    clock.tick(60) # the program will never run at more than 60 frames per second.
    # Be IDLE friendly
pygame.quit()
```



Vectors in Pygame

Homework : Draw components of the vector