video: https://youtu.be/Z0Tv-O7ibtY

3D Snake

This project started as an attempt at a rasterizer that could later be applied to produce a 3D game. A fundamental issue that I encountered is how pygame renders triangles: when rendering long triangles that stretch towards the camera, it is totally possible that one half of the triangle must be obscured while the other is visible. This would require the triangle to be drawn in two passes, something that pygame.draw.polygon is unable to do. After careful consideration I decided to build a 3D version of the classic game snake, as this is constructed from a uniform grid of squares. Changing this to cubes would lead to the edge case never occurring, as the triangles involved never pass through one another.

The design of the project is split into two main stages, the changes to the snake and the rendering of the board.

Moving the Snake
To create a trailing tail, I store the coordinates of each tail segment in a list and, every movestep, delete the last segment of the snake and create a new segment at the position where the snake head was last movestep. This creates the impression that the tail trails along such as in normal snake. One issue that arises with a camera that can move is that, intuitively, the WASD keys should change the direction of the snake relative to where the camera faces. To do this, we calculate which 90° sector the camera faces, and then index this into a key-specific list of directions that specifies which direction the user intends to go.

Rendering the board
The first step in creating a 3D grid of cubes is to understand the shape of a cube. To do this, I decode an object file of a cube exported from tinkercad. First we loop over the string that is the object file, sorting between vertices and triangles. Then we create a triangle object for each triangle in the cube and store those for later use. Then we find all the cubes stored in the 3D array that is the board, and use the stored triangle objects to create cubes at each relevant position.
Once we have all the triangles that make up the board, the next step is to render them. Each triangle object has a render method that finds the position of its vertices on the screen, as well as if the triangle faces towards the camera or not. We store these results in a list, and then sort the list by the distance of each triangle's midpoint to the camera. This is so that triangles in the foreground render on top of triangles in the background.

I am pleased with the outcome of the project: 3D snake is surprisingly challenging! However, were I to remake it, to increase frame rate I would render triangles on the GPU, and remove unnecessary triangles that are obscured either in the middle of or behind the snake’s tail.

Code:

Python

```
import numpy as np
import random
from sys import exit
import pygame # installs pygame
pygame.font.init()
font = pygame.font.SysFont('Nunito Sans', 40) # Grab a title font
small_font = pygame.font.SysFont('Nunito Sans', 28) # Grab a font for
flavour text
pygame.init() # initailises pygame
screen = pygame.display.set_mode((1000,700)) # creates a display surface
(window that the game runs in), stored in screen, must be passed both the
width and height at a minimium. If the code stops running, the window closes
pygame.display.set_caption("3D Snake") # changes the name of the window
clock = pygame.time.Clock() # creates a clock object used later to cap the
frame rate
FOCAL_LENGTH = 500
# 11 - Fast
# 6 - Normal
# 3 - Slow
FIXED_SPEED = 6 # This determines the speed of the snake. If set to 0 then
the snake accelerates after eating an apple
WIN_LENGTH = 40
speed = 6
frame = round(60/speed-1)
camera_coords = (0,0,0)
camera_yaw = 0
camera_pitch = 0
player_pos = (5,5,5)
apple_pos = (random.randint(0,11),random.randint(0,11),random.randint(0,11))
previous_player_pos = player_pos
player_dir = "left"
previous_mousecoords = (0,0)
ate_apple = True
apressed = 0
dpressed = 0
wpressed = 0
spressed = 0
qpressed = 0
epressed = 0
inputs = [0,0,0,0,0,0]
direction_held = False
dead = False
```

```
won = False
button_pressed = None
start_pressed = False
start_pressed_last_frame = False
grid = []
segments = []
# Create the 3d grid
for k in range(12):
   plane = []
   for o in range(12):
       minilist = []
       for i in range(12):
           minilist.append(None)
       plane.append(minilist.copy())
   grid.append(plane)


def get_side(a,b,p):
   lineAB = np.array((b[0]-a[0], b[1]-a[1]))
   lineAP = np.array((p[0]-a[0], p[1]-a[1]))
   rotateAB = np.array((-lineAB[1], lineAB[0]))
   return np.dot(lineAP,rotateAB) < 0


def rotate_yaw(x,y,z):
   global camera_yaw
   ihat = np.array([np.cos(camera_yaw),0,np.sin(camera_yaw)])
   jhat = np.array([0,1,0])
   khat = np.array([-np.sin(camera_yaw),0,np.cos(camera_yaw)])
   return x*ihat+y*jhat+z*khat


def rotate_pitch(vec):
   """
   Vertically rotates the np.array vec by camera_pitch.
   """
   global camera_pitch
   # Make the rotation matrix
   m = np.array([
       [1,0,0],
       [0,np.cos(camera_pitch), -np.sin(camera_pitch)],
       [0,np.sin(camera_pitch), np.cos(camera_pitch)]
   ])
   return m@vec


def world_to_screen(x,y,z,focal_length):
   """
```

```
   takes in the worldspace coords of a point and the focal length of the
camera, returns the screenspace coords of said point


   if the point is behind the screen, returns "Behind screen"
   """
   cache =
rotate_pitch(rotate_yaw(x-camera_coords[0],y-camera_coords[1],z-camera_coord
s[2]))
   x = cache[0] + camera_coords[0]
   y = cache[1] + camera_coords[1]
   z = cache[2] + camera_coords[2]
   if not z-camera_coords[2] <= 0.01:
       intersection_x = (focal_length * (x - camera_coords[0]))/(z -
camera_coords[2])+500
       intersection_y = (focal_length * (y - camera_coords[1]))/(z -
camera_coords[2])+350
       return((intersection_x,intersection_y), (x - camera_coords[0],y -
camera_coords[1],z - camera_coords[2]))
   else:
       return "Behind screen"


class triangle(): # store where a triangle is in the worldspace
   def __init__(self,p1,p2,p3,colour):
       self.p1 = p1
       self.p2 = p2
       self.p3 = p3
       self.colour = colour


   def render(self):
       p1_cache =
world_to_screen(self.p1[0],self.p1[1],self.p1[2],FOCAL_LENGTH)
       p2_cache =
world_to_screen(self.p2[0],self.p2[1],self.p2[2],FOCAL_LENGTH)
       p3_cache =
world_to_screen(self.p3[0],self.p3[1],self.p3[2],FOCAL_LENGTH)
       if p1_cache != "Behind screen" and p2_cache != "Behind screen" and
p3_cache != "Behind screen":
           middle = midpoint(p1_cache[0],p2_cache[0],p3_cache[0])
           p1_side = get_side(p1_cache[0],p2_cache[0],middle)
           p2_side = get_side(p2_cache[0],p3_cache[0],middle)
           p3_side = get_side(p3_cache[0],p1_cache[0],middle)
           if p1_side and p2_side and p3_side:
               p1_screenspace = p1_cache[0]
               p2_screenspace = p2_cache[0]
               p3_screenspace = p3_cache[0]
```

```
                return (self.colour, [p1_screenspace, p2_screenspace,
p3_screenspace]), (p1_cache[1],p2_cache[1],p3_cache[1])


# These are the vertexes and faces of a cube (imported from tinkerCAD)
imported_file = """
v 1         0       1
v 1         0       0
v 0         0       0
v 0         0       1
v 1         -1      1
v 1         -1      0
v 0         -1      0
v 0         -1      1


f 1     2   3 left
f 1     3   4 left
f 5     6   2 front
f 5     2   1 front
f 4     3   7 back
f 4     7   8 back
f 1     8   5 top
f 1     4   8 top
f 8     7   6 right
f 8     6   5 right
f 6     7   3 bottom
f 6     3   2 bottom
"""


vertexes = []
triangles_indexes = []
memory = ""
lines = imported_file.split("\n")
for line in lines:
   if line:
       if line[0] == "v":
           vertex = line.split()
           vertex = (float(vertex[1]),-float(vertex[3]),float(vertex[2])) #
Our axes are different to that of tinkerCAD so we need to convert them
           vertexes.append(vertex)
       if line[0] == "f":
           index = line.split()
           index = (index[1],index[2],index[3]) # Change the string into a
tuple of the infomation we use
           triangles_indexes.append(index) # Store the vertex indexes of the
triangles
```

```
box_triangles = []
for j,t in enumerate(triangles_indexes):
   # We then find the real position of these vertexes and use them to build
a triangle object
   v1 = vertexes[int(t[0])-1]
   v2 = vertexes[int(t[1])-1]
   v3 = vertexes[int(t[2])-1]
   box_triangles.append(triangle(v1,v2,v3,None))
   #
box_triangles.append(triangle(v1,v2,v3,["red","orange","yellow","blue","gree
n","purple"][round(np.floor(j/2))]))


def midpoint(p1, p2, p3):
   return ((p1[0]+p2[0]+p3[0])/3, (p1[1]+p2[1]+p3[1])/3)


def sort_function(hbt):
   distance = min(
       np.linalg.norm(hbt[1][0]),
       np.linalg.norm(hbt[1][1]),
       np.linalg.norm(hbt[1][2])
   )
   return distance







def draw_triangles():
   half_baked_triangles = []
   for t in triangles:
       render = t.render()
       if render:
           half_baked_triangles.append(render)
   half_baked_triangles.sort(key=sort_function,reverse=True)


   for hbt in half_baked_triangles:
       pygame.draw.polygon(screen,hbt[0][0],hbt[0][1])


def create_grid():
   for y,a in enumerate(grid):
       for x,b in enumerate(a):
           for z,c in enumerate(b):
               if c:
```

```
                    mesh = []
                    if z+1 <= 11:
                        if not grid[y][x][z+1]:
                            mesh.append(box_triangles[0])
                            mesh.append(box_triangles[1])
                    else:
                        mesh.append(box_triangles[0])
                        mesh.append(box_triangles[1])
                    if x+1 <= 11:
                        if not grid[y][x+1][z]:
                            mesh.append(box_triangles[2])
                            mesh.append(box_triangles[3])
                    else:
                        mesh.append(box_triangles[2])
                        mesh.append(box_triangles[3])
                    if x-1 >= 0:
                        if not grid[y][x-1][z]:
                            mesh.append(box_triangles[4])
                            mesh.append(box_triangles[5])
                    else:
                        mesh.append(box_triangles[4])
                        mesh.append(box_triangles[5])
                    if y-1 >= 0:
                        if not grid[y-1][x][z]:
                            mesh.append(box_triangles[6])
                            mesh.append(box_triangles[7])
                    else:
                        mesh.append(box_triangles[6])
                        mesh.append(box_triangles[7])
                    if z-1 >= 0:
                        if not grid[y][x][z-1]:
                            mesh.append(box_triangles[8])
                            mesh.append(box_triangles[9])
                    else:
                        mesh.append(box_triangles[8])
                        mesh.append(box_triangles[9])
                    if y+1 <= 11:
                        if not grid[y+1][x][z]:
                            mesh.append(box_triangles[10])
                            mesh.append(box_triangles[11])
                    else:
                        mesh.append(box_triangles[10])
                        mesh.append(box_triangles[11])
                    for t in mesh:

triangles.append(triangle((t.p1[0]+x,t.p1[1]+y,t.p1[2]+z),(t.p2[0]+x,t.p2[1]
+y,t.p2[2]+z),(t.p3[0]+x,t.p3[1]+y,t.p3[2]+z),c))
```

```
def set_square(x,y,z,thing):
   grid[y][x][z] = thing


def calculate_camera_position(center, distance, yaw, pitch):
   cx, cy, cz = center
   cam_x = cx - distance * np.cos(pitch) * np.sin(yaw)
   cam_y = cy - distance * np.sin(pitch)
   cam_z = cz - distance * np.cos(pitch) * np.cos(yaw)

   return (cam_x, cam_y, cam_z)


def player_movement():
    global player_dir, player_pos, grid, segments, dead, ate_apple,
apple_pos, offset, in_menu
    if ate_apple:
        # Add a new segment to the snake
        segments.append(previous_player_pos)
    # We then use offset to change which direction the snake would go by
indexing into a list of directions
    intended_dir = ""
    if wpressed:
            intended_dir = ["left","forward","right","backward"][(offset)]
    if apressed:
            intended_dir = ["backward","left","forward","right"][(offset)]
    if spressed:
            intended_dir = ["right","backward","left","forward"][(offset)]
    if dpressed:
            intended_dir = ["forward","right","backward","left"][(offset)]
    # We check that the player head cannot turn 180 degree into the snake
    if intended_dir == "left":
            if player_pos[2]+1 <= 11:
               if not
(player_pos[0],player_pos[1],player_pos[2]+1)==segments[0]:
                   player_dir = "left"
            else:
               if not (player_pos[0],player_pos[1],0)==segments[0]:
                   player_dir = "left"
    if intended_dir == "right":
        if player_pos[2]-1 >= 0:
            if not
(player_pos[0],player_pos[1],player_pos[2]-1)==segments[0]:
               player_dir = "right"
        else:
            if not (player_pos[0],player_pos[1],11)==segments[0]:
               player_dir = "right"
    if intended_dir == "forward":
```

```
        if player_pos[0]+1 <= 11:
            if not
(player_pos[0]+1,player_pos[1],player_pos[2])==segments[0]:
                player_dir = "forward"
        else:
            if not (0,player_pos[1],player_pos[2])==segments[0]:
                player_dir = "forward"
    if intended_dir == "backward":
        if player_pos[0]-1 >= 0:
            if not
(player_pos[0]-1,player_pos[1],player_pos[2])==segments[0]:
                 player_dir = "backward"
        else:
            if not (11,player_pos[1],player_pos[2])==segments[0]:
                player_dir = "backward"
    if qpressed:
        if player_pos[1]+1 <= 11:
            if not
(player_pos[0],player_pos[1]+1,player_pos[2])==segments[0]:
                player_dir = "down"
        else:
            if not ((player_pos[0],0,player_pos[2]))==segments[0]:
                player_dir = "down"
    if epressed:
        if player_pos[1]-1 >= 0:
            if not
(player_pos[0],player_pos[1]-1,player_pos[2])==segments[0]:
                player_dir = "up"
        else:
            if not (player_pos[0],11,player_pos[2])==segments[0]:
                player_dir = "up"

   # We nuke the grid, as we need to update the position of the snake
    grid = []
    for k in range(12):
        plane = []
        for o in range(12):
            minilist = []
            for i in range(12):
                minilist.append(None)
            plane.append(minilist.copy())
        grid.append(plane)
   # This checks if the snake head will go outside the cube, and if so wraps
around

    if player_dir == "forward":
        if player_pos[0]+1 <= 11:
            player_pos = (player_pos[0]+1,player_pos[1],player_pos[2])
```

```
        else:
            player_pos = (0,player_pos[1],player_pos[2])
    if player_dir == "backward":
        if player_pos[0]-1 >= 0:
            player_pos = (player_pos[0]-1,player_pos[1],player_pos[2])
        else:
            player_pos = (11,player_pos[1],player_pos[2])
    if player_dir == "up":
        if player_pos[1]-1 >= 0:
            player_pos = (player_pos[0],player_pos[1]-1,player_pos[2])
        else:
            player_pos = (player_pos[0],11,player_pos[2])
    if player_dir == "down":
        if player_pos[1]+1 <= 11:
            player_pos = (player_pos[0],player_pos[1]+1,player_pos[2])
        else:
            player_pos = (player_pos[0],0,player_pos[2])
    if player_dir == "left":
        if player_pos[2]+1 <= 11:
            player_pos = (player_pos[0],player_pos[1],player_pos[2]+1)
        else:
            player_pos = (player_pos[0],player_pos[1],0)
    if player_dir == "right":
        if player_pos[2]-1 >= 0:
            player_pos = (player_pos[0],player_pos[1],player_pos[2]-1)
        else:
            player_pos = (player_pos[0],player_pos[1],11)
     # If the snake eats itself, die
    if player_pos in segments:
        dead = True
        in_menu = True
   # If the snake ate an apple this frame, then move the apple
    if player_pos == apple_pos:
        ate_apple = True
        while apple_pos in segments or apple_pos == player_pos: # We ensure
that the apple does not spawn inside the snake
            apple_pos =
(random.randint(0,11),random.randint(0,11),random.randint(0,11))
    else:
        ate_apple = False
    # Move the snake
    if len(segments) != 0:
        # Remove the tail segment
        segments.remove(segments[-1])
        # Add a new segment to the front of the snake
        segments.insert(0,previous_player_pos)
    # Make alternating colours along the snake body
    for j,segment_coord in enumerate(segments):
```

```
set_square(segment_coord[0],segment_coord[1],segment_coord[2],["darkolivegre
en3","darkolivegreen4"][j%2])
    # Create the apple and player head
    set_square(apple_pos[0],apple_pos[1],apple_pos[2],"brown2")
    set_square(player_pos[0],player_pos[1],player_pos[2],"darkolivegreen1")

def create_rectangle(start_position,end_position,colour,border_radius):
  """Creates a rectangle centered on the position, with the given dimensions
and border radius. colour includes a fourth value in the tuple, that being
the alpha channel"""
  global screen, transparent_surface
  pygame.draw.rect(transparent_surface, colour,
(start_position[0],start_position[1],end_position[0]-start_position[0],end_p
osition[1]-start_position[1]),border_radius = border_radius)

def create_text(text,position,colour,f=font):
    global transparent_surface
    text_surface = f.render(text, True, colour)
    transparent_surface.blit(text_surface, position)
    return transparent_surface

def check_coords(start_position,end_position):
    global mousecoords
    x,y = mousecoords
    if x > start_position[0] and x < end_position[0] and y >
start_position[1] and y < end_position[1]:
        return True
    return False

# If the user presses an input (e.g. forward), the direction changes based
on which 90 degree sector the camera is looking into
# We calculate this at frame one, as sometimes we need it before it is
re-calculated in the function 'player_movement'
offset = round((camera_yaw%(2*np.pi))/(np.pi/2))%4
in_menu = True
while True:
    dt = clock.tick(clock.get_fps())/1000 # calculate deltatime, the time
spent rendering the last frame
    screen.fill("white") # Clear the screen
    for event in pygame.event.get(): # loops over all inputs detected by
pygame
        if event.type == pygame.QUIT: # if we try to close the window
            pygame.quit() # shuts down all of pygame
            exit() # Kills the while loop
    # This is the decides between the different gamemodes
    if FIXED_SPEED:
        speed = FIXED_SPEED
```

```
    else:
        speed = 0.210526 * len(segments) + 2.78947
    # We grab all the user input
    state = pygame.mouse.get_pressed()
    mousecoords = pygame.mouse.get_pos()
    keys = pygame.key.get_pressed()
    if in_menu:
        apressed,dpressed,wpressed,spressed,qpressed,epressed = 0,0,0,0,0,0
    else:
        apressed,dpressed,wpressed,spressed,qpressed,epressed =
keys[pygame.K_a],keys[pygame.K_d],keys[pygame.K_w],keys[pygame.K_s],keys[pyg
ame.K_q],keys[pygame.K_e]

    any_input = apressed or dpressed or wpressed or spressed or qpressed or
epressed
    # the WASD keys mean different things depending on which 90 degree
sector the camera is looking towards
    # offset is 0,1,2 or 3 depending on the sector
    if any_input:
        offset = round((camera_yaw%(2*np.pi))/(np.pi/2))%4
    mask = [apressed, dpressed, wpressed, spressed, qpressed, epressed]
    for i, m in enumerate(mask):
        # Increase everything that was pressed
        inputs[i] += m
        # Zero everything that wasn't pressed
        if any_input: # We need to check this, because otherwise short key
presses get reset before a move step
            inputs[i] *= m
    # Find the most recent input
    # This is the lowest value in inputs that is not zero
    lowest = None
    for j,k in enumerate(inputs):
        if lowest != None:
            if k < inputs[lowest] and k != 0:
                lowest = j
        else:
            if k != 0:
                lowest = j
    # We then zero all but the most recent input
    l = [0,0,0,0,0,0]
    if lowest != None:
        l[lowest] = 1
    # And then dictate that as the key being pressed
    apressed,dpressed,wpressed,spressed,qpressed,epressed = l
    frame += 100*dt
    if frame >= round(60/speed):
        frame = 0
        if not dead:
```

```
            player_movement()
    triangles = []
    create_grid()
    for t in box_triangles:
        scale = 12
        x_offset = 0
        y_offset = 11
        z_offset = 11
        p1 =
(t.p1[0]*scale+x_offset,t.p1[1]*scale+y_offset,t.p1[2]*scale+z_offset)
        p2 =
(t.p2[0]*scale+x_offset,t.p2[1]*scale+y_offset,t.p2[2]*scale+z_offset)
        p3 =
(t.p3[0]*scale+x_offset,t.p3[1]*scale+y_offset,t.p3[2]*scale+z_offset)
        a = world_to_screen(p1[0],p1[1],p1[2],FOCAL_LENGTH)
        b = world_to_screen(p2[0],p2[1],p2[2],FOCAL_LENGTH)
        c = world_to_screen(p3[0],p3[1],p3[2],FOCAL_LENGTH)
        ab = np.linalg.norm(np.array(p1)-np.array(p2))
        bc = np.linalg.norm(np.array(p2)-np.array(p3))
        ca = np.linalg.norm(np.array(p3)-np.array(p1))
        if a != "Behind screen" and b != "Behind screen" and c != "Behind
screen":
            if ab <= 12:
                pygame.draw.line(screen,"black",a[0],b[0],1)
            if bc <= 12:
                pygame.draw.line(screen,"black",b[0],c[0],1)
            if ca <= 12:
                pygame.draw.line(screen,"black",c[0],a[0],1)
    if state[0] and not in_menu or state[2] and in_menu:
        camera_yaw = camera_yaw + (mousecoords[0] - previous_mousecoords[0])
/ 100
        camera_pitch = camera_pitch + (mousecoords[1] -
previous_mousecoords[1]) / 100
    camera_coords =
calculate_camera_position((6,5,5),15,camera_yaw,camera_pitch)
    # draw all elements
    draw_triangles()
    # update everything
    previous_mousecoords = np.array(pygame.mouse.get_pos())
    previous_player_pos = player_pos
    # If the snake reaches a given length, they win!
    if len(segments) >= WIN_LENGTH and button_pressed == None:
        dead = True
        in_menu = True
        won = True
        button_pressed = None
    if in_menu:
        if not state[2]:
```

```
            camera_pitch += 0.05 - camera_pitch / 10
            camera_yaw += dt
        transparent_surface = pygame.Surface((1000, 700), pygame.SRCALPHA)
        exit_offset = (0,0)
        create_rectangle((250,200),(750,430),(0,0,0,125),10)
        if check_coords((667,210),(740,255)):
            if state[0]:
                pygame.quit() # shuts down all of pygame
                exit() # Kills the while loop
            else:
                exit_offset = (5,0)

create_rectangle((667+exit_offset[0],210+exit_offset[1]),(740+exit_offset[0]
,255+exit_offset[1]),(0,0,0,150),10)
                exit_offset = (0,-7)

create_rectangle((667+exit_offset[0],210+exit_offset[1]),(740+exit_offset[0]
,255+exit_offset[1]),(0,0,0,200),10)

create_text("Exit",(677+exit_offset[0],220+exit_offset[1]),(255,255,255))
        start_offset = (0,0)
        if check_coords((647,360),(740,420)):
            if state[0]:
                start_pressed = True
                start_offset = (2.5,0)

create_rectangle((647+start_offset[0],360+start_offset[1]),(740+start_offset
[0],420+start_offset[1]),(0,0,0,150),10)
                start_offset = (0,-3.5)
            else:
                start_pressed = False
                start_offset = (5,0)

create_rectangle((647+start_offset[0],360+start_offset[1]),(740+start_offset
[0],420+start_offset[1]),(0,0,0,150),10)
                start_offset = (0,-7)
            if start_pressed_last_frame and not start_pressed:
                in_menu = False
                grid = []
                segments = []
                # Create the 3d grid
                for k in range(12):
                    plane = []
                for o in range(12):
                    minilist = []
                    for i in range(12):
                        minilist.append(None)
                    plane.append(minilist.copy())
```

```
                grid.append(plane)
                dead = False
                ate_apple = True
                player_pos = (5,5,5)
                player_dir = "left"
                button_pressed = None
        else:
            start_pressed = False

create_rectangle((647+start_offset[0],360+start_offset[1]),(740+start_offset
[0],420+start_offset[1]),(0,0,0,200),10)

create_text("Start",(662+start_offset[0],375+start_offset[1]),(255,255,255))
        buttons = ["Slow","Medium","Fast","Classic"]
        for j,text in enumerate(buttons):
            button_offset = (0,0)
            flag = check_coords((260,210+j*55),(380,255+j*55))
            if flag and state[0]:
                    button_pressed = j
                    button_offset = (2.5,0)

create_rectangle((260+button_offset[0],210+j*55+button_offset[1]),(385+butto
n_offset[0],255+j*55+button_offset[1]),(0,0,0,150),10)
                    button_offset = (0,-3.5)
            if flag and not state[0] or j == button_pressed and not flag:
                button_offset = (5,0)

create_rectangle((260+button_offset[0],210+j*55+button_offset[1]),(385+butto
n_offset[0],255+j*55+button_offset[1]),(0,0,0,150),10)
                button_offset = (0,-7)

create_rectangle((260+button_offset[0],210+j*55+button_offset[1]),(385+butto
n_offset[0],255+j*55+button_offset[1]),(0,0,0,200),10)

create_text(text,(270+button_offset[0],220+j*55+button_offset[1]),(255,255,2
55))
        texts = [("An ideal pace to","learn the controls"),
                 ("The default pace","for casual games"),
                 ("A challenging pace","for the adventurous"),
                 ("Accelerates as the","game progresses")]
        speeds = [3,6,11,0]
        if button_pressed != None:
            won = False
        if won:
            create_text("Victory!",(430,220),(255,255,255))
        if button_pressed != None:
            # Set the speed to the users decision
            FIXED_SPEED = speeds[button_pressed]
```

```
            if not button_pressed_last_frame:
                frame = round(60/speed-1)
            if not won:
                create_text(buttons[button_pressed],(430,220),(255,255,255))

create_text(texts[button_pressed][0],(430,260),(255,255,255),small_font)

create_text(texts[button_pressed][1],(430,285),(255,255,255),small_font)
        screen.blit(transparent_surface, (0, 0))
    if button_pressed != None:
        button_pressed_last_frame = True
    else:
        button_pressed_last_frame = False
    start_pressed_last_frame = start_pressed
    pygame.display.update() # updates the display surface
    clock.tick(60) # stops the while loop from running more than 60 times a
second.
```