Izrada jednostavne arkadne igrice u Pythonu

Crljenko, Jerko

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:385336

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Image not found or type unknown

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

zir.nsk.hr

Image not found or type unknown

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

JERKO CRLJENKO

IZRADA JEDNOSTAVNE ARKADNE IGRICE U PYTHONU

Završni rad

Pula, rujan 2018. godine

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

JERKO CRLJENKO

IZRADA JEDNOSTAVNE ARKADNE IGRICE U PYTHONU

Završni rad

JMBAG: 0303038769, izvanredni student

Studijski smjer: Informatika

Predmet: Programiranje

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijsko i programsko inženjerstvo

Mentor: doc. dr. sc. Krunoslav Puljić

Pula, rujan 2018. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Jerko Crljenko, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, 2018. godine

IZJAVA

o korištenju autorskog djela

Ja, Jerko Crljenko dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom Izrada jednostavne arkadne igrice u Pythonu koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

Potpis

_____

U Puli, _____, 2018. godine

SADRŽAJ

1. UVOD 1

2. PROGRAMSKI JEZIK PYTHON 2

3. PAKET PYGAME 4

4. IZRADA KOSTURA IGRE *INVADERS* 5

4.1. Kostur igre, proceduralno 5

4.2. Kostur igre, klasa Game 10

5. IZRADA LETJELICE IGRAČA 13

5.1. Prikaz letjelice, klasa Player 13

5.2. Pomicanje letjelice, move() 15

6. IZRADA NEPRIJATELJSKE VOJSKE, *INVADERA* 17

6.1. Prikaz invadera 17

6.2. Prikaz grupe invadera 17

6.3. Animacija invadera 18

6.4. Prikaz grupe različitih invadera 20

6.5. Vodoravno pomicanje invadera 21

6.6. Silazno pomicanje invadera 23

7. RAKETE I PUCANJE 24

7.1. Pucanje letjelice igrača 24

7.2. Kolizije rakete i invadera 25

7.3. Pucanje invadera 27

7.4. Pucanje samo najnižih invadera po stupcima 28

7.5. Kolizije rakete i letjelice, spritesheet 29

8. IZRADA BONUS LETJELICE, MYSTERY 32

8.1. Prikaz mystery letjelice 32

8.2. Pomicanje mystery letjelice s lijeva na desno 32

8.3. Pomicanje mystery letjelice u oba smjera i kolizija s raketom 33
9. TEKST, REZULTAT I ŽIVOTI IGRAČA, KRAJ IGRE 35
9.1. Prikaz statičnog teksta 35
9.2. Prikaz rezultata, dinamički tekst 35
9.3. Prikaz trepćućeg teksta 37
9.4. Prikaz života igrača 38
9.5. Kraj igre (game over) 39
9.6. Ponovno igranje nakon kraja igre, replay 40
9.7. Nagradni život, extra life 42
10. BUNKERI I ZVUKOVI 43
10.1. Izrada bunkera 43
10.2. Dodavanje zvukova 45
11. NAMJEŠTANJE TEŽINE I STANJA IGRE, INSTALACIJA 48
11.1. Podešavanje težine igre 48
11.2. Podešavanje stanja igre 49
11.3. Instalacija igre 52
12. ZAKLJUČAK 53
13. LITERATURA 54
WEB STRANICE 54
PYGAME I PYTHON DOKUMENTACIJA 54
IZVORI 55
POPIS SLIKA I TABLICA 56
SAŽETAK 57
SUMMARY 58

1. UVOD

U radu ćemo prikazati postupak nastajanja jednostavne arkadne igre rađene u Pythonu (verzija 3.7.0). Za potrebe rada izradili smo igru Invaders koristeći paket Pygame (verzija 1.9.4). Izvorni kod je dostupan na repozitoriju Github: https://github.com/jcrljenko/Invaders.

Cilj rada je prikazati postupak nastajanja igre, odnosno postupno građenje svakog elementa igre. Igra je nastajala u nekoliko faza, te ćemo svaku detaljno opisati i obrazložiti.

Krenut ćemo od izgradnje osnovnog kostura igre, tj. grafičkog prikaza prozora s pozadinskom slikom. Postupno ćemo uvoditi i dodavati nove elemente (letjelicu igrača, neprijateljsku vojsku *invadere*, rakete, bunkere…) sve do potpuno izgrađene i funkcionalne igre. Upozorit ćemo na mogućnosti i ograničenja koja se javljaju prilikom programiranja navedene igre.

No, prije samog prikaza postupka nastanka igre osvrnut ćemo se kratko na programski jezik Python te paket za izradu igara Pygame.

2. PROGRAMSKI JEZIK PYTHON

Python je viši programski jezik opće namjene. Godine 1991. razvio ga je Nizozemac Guido van Rossum, a ime mu je dao po kultnoj britanskoj seriji „Leteći cirkus Montyja Pythona“ (*Monty Python's Flying Circus*). U Python zajednici Rossum nosi nadimak *BDFL*, što je kratica za *Benevolent Dictator for Life* (dobroćudni doživotni diktator).[1]

Godine 2000. izdan je Python 2.0, a 2008. Python 3.0. Posljednja je verzija unijela brojne i značajne promjene u sintaksi jezika, te privlači sve više programera.

Python je zamišljen tako da ga mogu koristiti i ljudi koji nemaju još puno iskustva u programiranju. Sintaksa je vrlo čitka, a postoje i brojne knjige i upute diljem Interneta o njegovoj primjeni. Vrlo često će za pisanje nekog programa u Pythonu trebati puno manje koda nego u tradicionalnim programskim jezicima poput C++ ili Jave.[2]

Python podržava proceduralno, objektno orijentirano i funkcionalno programiranje. Iz tog razloga učenjem Pythona svladavaju se uz osnovne koncepte programiranja i paradigme različitih drugih programskih jezika.

Python je interpreterski jezik. Zbog toga izvođenje koda programa zna biti višestruko sporije nego kod programa pisanih u C ili C++, gdje se kod prevodi u strojni jezik i u tom obliku prosljeđuje korisniku. U Pythonu se program prevodi u Python *bytecode* kojega pokreće Python virtualna mašina.[3]

Python danas koriste brojne uspješne kompanije poput Google-a, IBM-a, Disney-a ili EA Games. Jedan je od najbrže rastućih jezika[4] i prema posljednjim podacima među tri najkorištenija programska jezika.[5]

Python je otvorenog koda (*open source*). Zbog toga se brojni paketi i moduli mogu besplatno nabaviti i skinuti na stranicama repozitorija PyPI (*Python Package*

[1] Paul Barry, *Head First Python*, 2nd Edition, Sebastopol: O’Reilly Media, 2017., 564.
[2] Michael Urban, Joel Murach, *Murach's Python Programming*, Fresno: Mike Murach & Associates, 2016., 4.-5.
[3] https://opensource.com/article/18/4/introduction-python-bytecode
[4] https://stackoverflow.blog/2017/09/06/incredible-growth-python/
[5] https://www.tiobe.com/tiobe-index/

Index).[6] U trenutku pisanja ovog rada na repozitoriju je bilo dostupno preko 150.000 projekata.

Za razliku od mnogih jezika Python nema deklaracije varijabli. One se jednostavno definiraju tako što im se dodijeli neka vrijednost. Jednako tako za razliku od većine jezika nema sintaksu s početkom ili krajem bloka. Svi se međusobni odnosi zasnivaju na „uvlačenju“ koda (*indentation*).

U Pythonu se mogu graditi i CLI (*command line interface*) i GUI (*graphic user interface*) aplikacije. Međutim, u njemu se ne mogu generirati samostalne izvršne datoteke (*binary*) iz skripta. Bez obzira na to, postoje paketi poput py2exe, PyInstaller ili cx_Freeze koje to omogućavaju.

Python dolazi već ugrađen i instaliran uz većinu Linux distribucija. Za operativni sustav Windows treba ga skinuti i instalirati. Instalacijske datoteke su dostupne na poveznici: https://www.python.org/downloads/release/python-370/. Prilikom instalacije predlažemo uključivanje Pythona u PATH varijablu okruženja (*Add Python to PATH*). Putem prilagođene instalacije (*Customize instalation*) u naprednim opcijama predlažemo dodavanje instalacije za sve korisnike (*Install for all users*).

[6] https://pypi.org/

3. PAKET PYGAME

Paket Pygame je skup modula za izgradnju jednostavnijih igara s grafičkim sučeljem. Izgrađen je na skupu biblioteka SDL (Simple DirectMedia Layer)[7] pomoću kojih pristupamo multimedijalnom hardveru računala. Pygame je kompatibilan s većinom operacijskih sustava i platformi (Linux, Windows, MacOS, …). Dijelovi Pygamea su pisani u programskom jeziku C, koji je nekoliko puta brži od Pythona.[8]

Pygame nam olakšava crtanje slika, ilustracija i animacija na zaslonu, jednostavno rukovanje događajima (*events*) kao što su pritisak tipke tipkovnice, pokret miša ili igraće palice, učitavanje i reprodukciju zvuka, kolizije ili sudare objekata.

NAZIV MODULA	NAMJENA
`pygame.display`	upravljanje prikaza prozora ili zaslona
`pygame.draw`	crtanje jednostavnih oblika (pravokutnici, linije, mnogokuti…)
`pygame.event`	interakcija s događajima (tipkovnica, miš, joystick…)
`pygame.font`	učitavanje i renderiranje fontova
`pygame.image`	učitavanje i snimanje slika
`pygame.key`	upravljanje tipkovnicom
`pygame.mixer`	učitavanje i reprodukcija zvukova
`pygame.mouse`	upravljanje mišem
`pygame.sprite`	skup klasa za objekte igre (Sprite, Group…), funkcije za koliziju
`pygame.time`	nadgledanje, kontrola vremena
`pygame.transform`	manipulacija površinom (promjena veličine, rotacija, okretanje, filtriranje slika)

Tablica 1: Popis važnijih Pygame modula

Pygame je otvorenog koda (open source), besplatan je i izdan pod GPL licencom. Ne dolazi s Pythonom, te ga je za korištenje potrebno dodatno instalirati.

Na računalima s Windows operativnim sustavom postoji više načina njegove instalacije, a najjednostavnija bi bila putem *command prompta*, u kojeg treba upisati: `pip install pygame==1.9.4`. Ako Python nije instaliran za sve korisnike, onda *command prompt* treba pokrenuti kao administrator i upisati istu naredbu.

[7] https://www.libsdl.org/
[8] https://www.pygame.org/docs/tut/PygameIntro.html

4. IZRADA KOSTURA IGRE *INVADERS*

4.1. Kostur igre, proceduralno

Najprije ćemo izraditi prozor s prikazom pozadinske slike. Na početku programa uključujemo (uvozimo) module i pakete čije ćemo dijelove koristiti u izradi programa.

Moduli su dijelovi softwarea, specifične funkcionalnosti, unutar kojih su smještene definicije srodnih klasa, funkcija ili konstanti. Modul je datoteka s *.py nastavkom (ekstenzijom), a ime datoteke ujedno predstavlja i ime modula. Instalacijom Pythona dobivamo i brojne standardne module, kao primjerice `random` (modul za generiranje pseudoslučajnih brojeva) ili `math` (modul za brojne matematičke funkcije).[9]

Paketi su skupovi modula smješteni unutar iste mape (eng. *folder*). Kao i kod modula, ime mape je ujedno i ime paketa.

Uključivanje modula i paketa izvodimo na nekoliko načina, ovisno o tome želimo li uključiti cijeli modul odnosno paket ili samo neke određene dijelove.

```
import <paket_ili_modul>
import <paket.modul.objekt>
import <paket_ili_modul> as <promijenjeni_naziv_ili_kratica>
from <paket> import <modul_ili_podpaket_ili_objekt>
from <modul> import <objekt>
from <modul> import *
```

U izradi naše arkadne igre koristit ćemo paket Pygame.

```
# Uključivanje objekta environ iz modula os
from os import environ

# Uključivanje Pygame paketa, te njegovo imenovanje kao pg
import pygame as pg
# Uključivanje svih Pygame konstanti
from pygame.locals import *
```

Uz uključivanje Pygame paketa i njegovih konstanti, uključili smo i modul `os`. On omogućava korištenje funkcija za interakciju s operativnim sustavom. Pristup varijablama okruženja (eng. *environment variables*) omogućava `environ`, pridruženi objekt (eng. *mapping object*) modula `os`.[10] Varijable okruženja pohranjuju informacije

[9] Zoran Kalafatić, Antonio Pošćić, Siniša Šegvić, Julijan Šribar, Python za znatiželjne. Sasvim drukčiji pogled na programiranje, Zagreb: Element, 2016., 291.
[10] https://docs.python.org/3/library/os.html

o operativnom sustavu i korisničkim podacima. Koristeći varijable okruženja, operativnom sustavu možemo prenijeti određene potrebne informacije.

Budući da je Pygame zasnovan na SDL biblioteci, iskoristit ćemo SDL varijablu za početno pozicioniranje prozora igre. SDL_VIDEO_WINDOW_POS varijabli okruženja dodjeljujemo dvije cjelobrojne vrijednosti u obliku znakovnog niza, tj. *stringa*. Dodijeljena pozicija je gornji lijevi kut prozora unutar koordinata zaslona.[11]

```
environ["SDL_VIDEO_WINDOW_POS"] = "100, 100"
```

Definirat ćemo i konstante za veličinu, širinu i visinu prozora, kao i konstantu za broj prikaza u sekundi (eng. *frames per second*).

```
# Konstante igre:
# Veličina, širina i visina prozora
SCREEN_SIZE = SCREEN_WIDTH, SCREEN_HEIGHT = (800, 600)
# Broj frameova u sekundi
FPS = 60
```

Veličini prozora (`SCREEN_SIZE`) dodjeljujemo n-torku (eng. *tuple*) (`800, 600`). N-torke su sekvence zarezom razdvojenih elemenata. Svakom se elementu može pristupati putem indeksa. Mogu sadržavati objekte različitih tipova. N-torke su neizmjenljive, što znači da nije moguće dodavati ili brisati elemente, kao ni mijenjati poredak elemenata.[12]

```
>>> # Različiti tipovi u n-torci: string, int, float, bool
>>> student = ("Ivo Ivic", 25, 4.35, True)
>>>
>>> # Pristup elementima n-torke putem indeksa.
>>> student[0]
'Ivo Ivic'
>>> student[2]
4.35
>>> # Pristup zadnjem elementu sekvence
>>> student[-1]
True
```

Pojedine elemente n-torke moguće je „raspakirati“ u slijed varijabli, pa smo uz veličinu prozora odmah odredili širinu prozora (`SCREEN_WIDTH`) na 800 i visinu prozora (`SCREEN_HEIGHT`) na 600.

[11] https://www.pygame.org/wiki/FrequentlyAskedQuestions
[12] Z. Kalafatić i dr., *Python za znatiželjne*, 105.

```
>>> # Svaki element n-torke dodijeljen je jednoj varijabli
>>> ime_prezime, godine, prosjek, uvjet_polozen = student
>>> ime_prezime
'Ivo Ivic'
>>> godine
25
>>> prosjek
4.35
>>> uvjet_polozen
True
```

Nakon što smo uključili module i pakete, te definirali konstante, definirat ćemo glavnu funkciju u kojoj ćemo namjestiti postavke prozora i u kojoj će se izvoditi glavna petlja. Prije nego počnemo rad s Pygame paketom, potrebno je inicijalizirati njegove module. To se izvodi pozivom `pygame.init()` koji nam omogućuje automatsku inicijalizaciju svih potrebnih modula.[13] U igri ćemo koristiti sljedeće Pygame module: `display`, `draw`, `event`, `font`, `image`, `key`, `time` i `sprite`.

```
def main():
    # Inicijalizacija svih Pygame modula
    pg.init()
```

Postavit ćemo prozor na zadanu veličinu, navesti naslov, te onemogućiti prikaz strelice miša unutar prozora igre.

```
# Postavljanje prozora na zadanu veličinu
screen = pg.display.set_mode(SCREEN_SIZE)

# Postavljanje naslova prozora i onemogućavanje prikaza miša
pg.display.set_caption("Invaders")
pg.mouse.set_visible(False)
```

Ranije pripremljenu sliku[14] učitat ćemo koristeći funkciju `load()` iz modula `image`. Argument funkcije je ime slike i mapa u kojoj je slika pohranjena u obliku *stringa*.[15] Pygame zatim kreira `Surface` objekt koji služi za prikaz svih slika.[16] Budući da Python omogućava povezivanje više funkcija i/ili metoda u nizu, nakon funkcije `load()` i kreiranja `Surface` objekta, na njemu odmah pozivamo metodu `convert()` koja bitno ubrzava crtanje učitane slike.[17]

[13] https://www.pygame.org/docs/tut/ImportInit.html
[14] Izvor slike: https://www.allwallpaper.in/outer-space-stars-wallpaper-16838.html
[15] https://www.pygame.org/docs/ref/image.html
[16] https://www.pygame.org/docs/ref/surface.html
[17] https://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert

```
# Učitavanje i konverzija slike pozadine
bg_image = pg.image.load("images/background.jpg").convert()
```

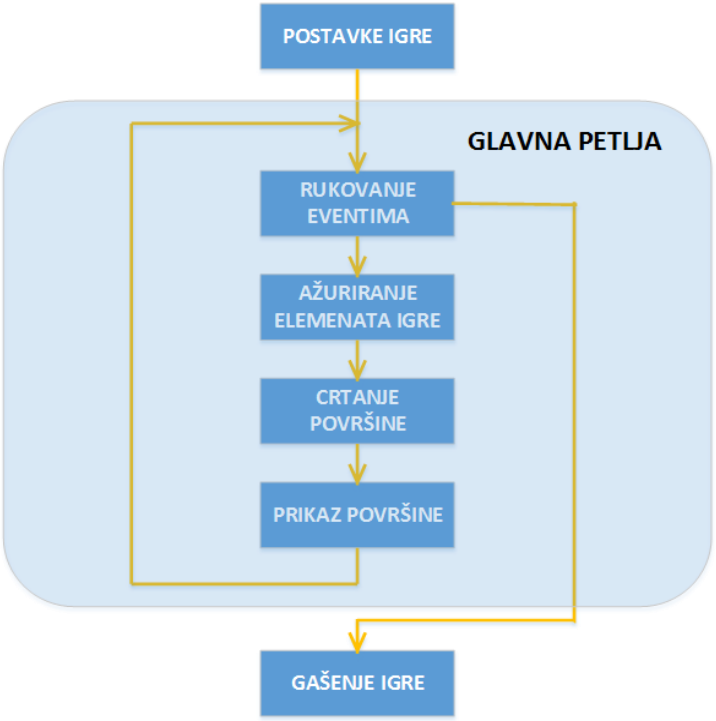
Stvorit ćemo „sat“ s kojim ćemo upravljati brzinom osvježavanja prozora. *Boolean* varijabla done omogućit će nam da možemo u bilo kojem trenutku zaustaviti izvođenje glavne petlje.

```
# Stvaranje sata kojim ćemo upravljati osvježavanje prozora.
clock = pg.time.Clock()

# Boolean varijabla za kontrolu izvršavanja petlje.
done = False
```

Središnji dio igre predstavlja glavna petlja. U njezinom se dijelu kontinuirano ponavljaju iste radnje:

- ispitivanje i rukovanje događajima (eng. *events)* – pita se sustav je li se neki događaj desio i odgovara se na odgovarajući način,
- ažuriranje podataka, objekata i elemenata igre,
- crtanje trenutnog stanja igre na nevidljivoj površini (eng. *surface*),
- prikaz upravo nacrtane površine.[18]



Slika 1: Glavna programska petlja

[18] Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers, How to Think Like a Computer Scientist: Learning with Python 3 Documentation, 3rd Edition, 2012. (korišteno prema: http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/ , zadnji posjet 4. 9. 2018.), 228.

Na početku glavne programske petlje u `for` petlji propitujemo događaje. Događaji mogu biti dodir tipke na tipkovnici, pritisak gumba miša, pomicanje palice za igranje ili joysticka. Provjeravat ćemo je li pritisnuta tipka *Escape* na tipkovnici ili pritisnut gumb *Close* za zatvaranje prozora. U slučaju da se desio neki od događaja prekida se izvođenje petlje, tj. prekida se izvođenje programa.

```
while not done:
    # Petlja za obradu eventova (event processing loop)
    for event in pg.event.get():
        if event.type == QUIT or
                        event.type == KEYDOWN and event.key == K_ESCAPE:
            done = True

        # Ispunjavanje površine slikom
        screen.blit(bg_image, (0, 0))

        # Ažuriranje prozora svime što smo nacrtali na površini.
        pg.display.flip()

        # Ograničavanje broja frameova u sekundi
        clock.tick(FPS)

    # Izlazak iz programa
    pg.quit()
```

U trenutnom razvoju igre nemamo još elemenata za ažuriranje, pa stoga preskačemo taj korak i crtamo površinu. U svakoj iteraciji petlje, tj. u svakom *frameu* crta se sve od nule. Zato moramo crtati i pozadinsku sliku svaki *frame*. Za tu potrebu koristimo metodu `blit()` klase `Surface`. Pri pozivu metode šaljemo dva argumenta: učitanu sliku (`bg_image`) i odredište (eng. *destination*) koje predstavlja n-torka koordinata lijevog gornjeg kuta slike – (`0, 0`).[19]

Kad je „nevidljiva“ površina pripremljena, vrijeme je za njezin prikaz u prozoru. Prikaz u prozoru svega nacrtanog na površini omogućava funkcija `flip()` modula `display`.[20]

Na kraju petlje još samo kontroliramo koliko će se *frameova* prikazivati svake sekunde. Metoda `tick()` se poziva jednom po *frameu* i izračunava koliko je milisekundi prošlo od prethodnog poziva. Pozivom `clock.tick(60)` jednom po *frameu*, program će raditi na 60 *frameova* u sekundi.[21]

[19] https://www.pygame.org/docs/ref/surface.html#pygame.Surface.blit
[20] https://www.pygame.org/docs/ref/display.html#pygame.display.flip
[21] https://www.pygame.org/docs/ref/time.html#pygame.time.Clock.tick

Pomoću `pygame.quit()` deinicijalizirat ćemo sve module Pygamea koji su na početku inicijalizirani.

Na kraju još samo treba pokrenuti izvođenje glavne funkcije `main()`. Prvo provjeravamo tzv. ispitnim kodom je li modul izravno pokrenut kao skripta, tj. glavni program ili je kako smo već ranije napomenuli uključen naredbom `import`. Prilikom pokretanja programa, Python postavlja brojne ugrađene (eng. *built-in*) varijable između kojih i varijablu `__name__`. Ako je modul pokrenut kao skripta ili glavni program vrijednost varijable `__name__` bit će postavljena kao `"__main__"`. U suprotnom, ako je modul pokrenut zahvaljujući `import` naredbi, vrijednost varijable `__name__` bit će postavljena na ime datoteke, odnosno modula.[22]

Budući da modul pokrećemo kao glavni program pokrenut će se funkcija `main()`.

```
# Pozovi main funkciju ako je ova skripta pokrenuta,
# tj. nije uvezena (import).
if __name__ == "__main__":
    main()
```

4.2. Kostur igre, klasa Game

Nakon što smo kreirali osnovni kostur proceduralno, nastavljamo izradu igre objektno orijentiranim pristupom. Svaki će objekt igre postati posebna, zatvorena cjelina. Neki će objekti biti i vrlo brojni, kao npr. neprijateljski *invaderi*. Objekte ćemo lakše kreirati i ažurirati ako su svi njihovi atributi i metode na jednom mjestu.

Osnovna klasa igre je klasa `Game`. Tu ćemo pokretati igru, učitavati slike i zvukove, prikazivati letjelicu igrača, neprijateljske letjelice, rakete, pratiti je li došlo do kolizije (eng. *collision*), itd.

Klasa `Game` predstavljat će jednu instancu igre. Kad je igrač na početku pokrene, bit će aktivna sve do kraja te igre. Ako igrač poželi odigrati još jednu igru, jednostavno će se kreirati nova instanca klase i nova igra može započeti.

Prilikom kreiranja objekata klase prvo se izvodi metoda `__init__`, odnosno konstruktor. U njemu kreiramo sve potrebne atribute i inicijaliziramo igru. Svaki atribut

[22] Steven F. Lott, Modern Python Cookbook, Birmingham: Packt Publishing Ltd., 2016., 151.

uz svoje ime mora imati i riječ `self`, npr. `self.image.` I metodama klase se kod definiranja dodaje prvo parametar `self`, a onda ostali parametri, ako ih metoda ima.[23]

```
class Game:
    def __init__(self):
        self.images = {}
        self.load_images()

    def load_images(self):
        self.images["background"] =
                        pg.image.load("images/background.jpg").convert()

    def process_events(self):
        for event in pg.event.get():
            if event.type == QUIT or
                        event.type == KEYDOWN and event.key == K_ESCAPE:
                return True

    def display_frame(self, screen):
        screen.blit(self.images["background"], (0, 0))
        pg.display.flip()
```

Konstruktor naše klase `Game` će zasad samo učitavati pozadinsku sliku. Budući da će u igri biti puno slika kreirat ćemo posebnu metodu za učitavanje slika `load_images()`. Sve će se slike učitavati u rječnik (eng. *dictionary*).

Za razliku od n-torki elemente rječnika ne možemo indeksirati. Svaki će element rječnika imati ključ (*key*) i njemu pripadnu vrijednost (*value*). Elementi rječnika se navode unutar vitičastih zagrada, odvajaju se zarezom, dok se ključ od vrijednosti razdvaja dvotočkom.[24]

```
>>> # Rjecnik student
>>> # Kljucevi: "ime", "prezime", "godine", ...
>>> # Vrijednosti: "Ivo", "Ivic", 25, ...
>>> student = {
      "ime": "Ivo",
      "prezime": "Ivic",
      "godine": 25,
      "prosjek": 4.35,
      "uvjet": True
      }
>>> student["ime"]
'Ivo'
>>> student["uvjet"]
True
```

[23] Leo Budin, Predrag Brođanac, Zlatka Markučič, Smiljana Perić, *Napredno rješavanje problema programiranjem u Pythonu*, Zagreb: Element, 2013., 5.
[24] Leo Budin, Predrag Brođanac, Zlatka Markučič, Smiljana Perić, *Rješavanje problema programiranjem u Pythonu*, Zagreb: Element, 2014., 222.

Obrada svih događaja (*events*) bit će u posebnoj metodi `process_events()`. Metoda će se pozivati u glavnoj petlji `main()` funkcije. Na sličan način kao i prije metoda će vratiti `True` ako je pritisnuta tipka *Escape* ili je pokrenuto zatvaranje prozora.

Metoda `display_frame()` prikazivat će pozadinsku sliku.

Na kraju je potrebno još samo kreirati instancu klase Game: `game = Game()`

5. IZRADA LETJELICE IGRAČA

5.1. Prikaz letjelice, klasa Player

Pomoću klase `Player` izgradit ćemo letjelicu. Klasa `Player`, kao i većina klasa u igri, naslijedit će klasu `Sprite` (modula `sprite`).[25] Nasljeđivanje će nam omogućiti iskorištavanje već postojećih metoda i atributa klase `Sprite`, a to će bitno pojednostaviti ažuriranje i crtanje svih *spriteova* na zaslonu. Uz naslijeđene metode i atribute podklasama ćemo dodati još neke metode i atribute.

```
class Player(pg.sprite.Sprite):
    # Početna pozicija Playera.
    position = (400, 550)

    def __init__(self, image):
        super().__init__()
        self.size = (50, 25)
        self.image = pg.transform.scale(image, self.size)
        self.rect = self.image.get_rect(center=self.position)
```

Atribut `position` je klasni ili statički, tj. pripada cijeloj klasi, a ne samo njezinim pojedinačnim instancama. Klasni atributi postoje neovisno o instancama klase, te ih možemo dohvaćati i prije nego kreiramo ijedan objekt klase.[26]

Unutar konstruktora svake podklase koristeći metodu `super()` trebamo pozvati konstruktor nadklase kako bi se naslijedili eventualni atributi nadklase i izbjeglo dupliciranje koda: `super().__init__()`.[27]

Konstruktor klase `Player` ima dodatni parametar `image` pomoću kojeg primamo sliku letjelice igrača, koju ćemo transformirati na zadanu veličinu, tj. *skalirati*. Također definiramo i pravokutni okvir `rect` koji nam omogućava spremanje pravokutnih koordinata slike na zadanu poziciju: `Rect(left, top, width, height)`.



Slika 2: Player

[25] Sprite je dvodimenzionalna slika koja se može kretati po zaslonu neovisno o drugim slikama. Sprite može biti lik ili karakter u igri, neprijateljska letjelica ili samo pozadinska slika.
[26] Z. Kalafatić i dr., *Python za znatiželjne*, 342.
[27] Dusty,Phillips, *Python 3 Object-oriented Programming*, 2nd Edition, Birmingham: Packt Publishing Ltd., 2015, 63.-64.

Za spremanje i upravljanje s više *sprite* objekata unutar Pygame paketa postoji posebna kontejnerska klasa `Group`. Za *playera* ćemo koristiti posebnu grupu: `GroupSingle`, u koju možemo spremiti samo jedan *sprite*. Korištenje grupe će nam kasnije omogućiti kontrolu ispaljivanja raketa ili lakšu provjeru kolizija.[28]

Atribut `all_sprites` je kontejner u kojeg ćemo pohranjivati sve *spriteove* u igri. To će nam omogućiti da jednostavno ažuriramo i crtamo *spriteove* u glavnoj petlji.

```
self.all_sprites = pg.sprite.Group()
self.player_grp = pg.sprite.GroupSingle()
```

Metodu `load_images()` ćemo preinačiti tako da se učitavaju sve slike potrebne u igri. U rječnik se učita i konvertira slika, tako da njezin naziv predstavlja *key* rječnika, a sama učitana slika *value* rječnika. Ujedno se postavlja i transparentnost svih slika metodom `set_colorkey()`. Budući da je pozadinska slika jedina u JPG formatu i ne treba postavljanje transparentnosti, ona će se jedina posebno učitavati. Sve će se slike pohranjivati u listi slika, a pri učitavanju slika koristimo *f-string*, tj. formatirani *string*.[29]

```
def load_images(self):
    image_names = ["player"]
    for img in image_names:
        self.images[img] = pg.image.load(f"images/{img}.png").convert()
        self.images[img].set_colorkey(pg.Color("Black"))

    self.images["background"] =
                pg.image.load("images/background.jpg").convert()
```

Liste su najčešće korištene sekvence u Pythonu. Slično kao i n-torke, elementi liste su indeksirani i mogu biti različitih tipova. Za razliku od n-torki liste su izmjenljivi objekti, što nam omogućava dodavanje i brisanje elemenata, kao i mijenjanje poretka elemenata.[30]

Igrača instanciramo u metodi `create_player()` klase `Game`, te ga odmah dodajemo u grupe:

```
def create_player(self):
    self.player = Player(self.images["player"])
    self.player_grp.add(self.player)
    self.all_sprites.add(self.player)
```

[28] https://www.pygame.org/docs/ref/sprite.html
[29] f-string, tj. formatirani string dostupan je od Pythona verzije 3.6
[30] Z. Kalafatić i dr., *Python za znatiželjne*, 111. – 112.

Na kraju će još samo u metodi `display_frame()`, koja se izvodi svaki *frame*, grupa `all_sprites` pozivati naslijeđenu metodu `draw()`, te će igrač biti ucrtan kad pokrenemo program: `self.all_sprites.draw(screen)`.

5.2. Pomicanje letjelice, move()

Kako bi pokrenuli letjelicu, prvo moramo dodati atribute `move_x` i `bounds`. Pomoću `move_x` atributa kontroliramo pomicanje letjelice ulijevo i udesno za 5 piksela po *frameu*. Atributom `bounds` dodajemo granice prostora kretanja letjelice.

```
# lijevo <-> desno za 5 pixela po frameu
self.move_x = 5
self.bounds = pg.Rect(10, 538, 780, 25)
```

Pomoću nadjačane metode `update()` kontrolirat ćemo pokret letjelice (nadjačava se istoimena metoda naslijeđene klase `Sprite`). Parametar `keys` predstavlja pritisnute tipke. Ako je pritisnuta lijeva ili desna tipka tipkovnice, letjelica se pomiče ulijevo, odnosno udesno. Parametar `*args` omogućava slanje varijabilnog broja argumenata metodi `update()`. Iako trenutnoj metodi `update()` nisu potrebni dodatni parametri, to je nužno dodati jer će i druge klase, koje tek trebamo napisati, naslijediti klasu `Sprite`, a koristit će neke druge parametre za `update()` metodu. Sve će te klase imati nadjačane metode `update()` s različitim parametrima.

```
def update(self, keys, *args):
    if keys[K_LEFT]:
        self.rect.move_ip(-self.move_x, 0)
    if keys[K_RIGHT]:
        self.rect.move_ip(self.move_x, 0)

    self.rect.clamp_ip(self.bounds)
```

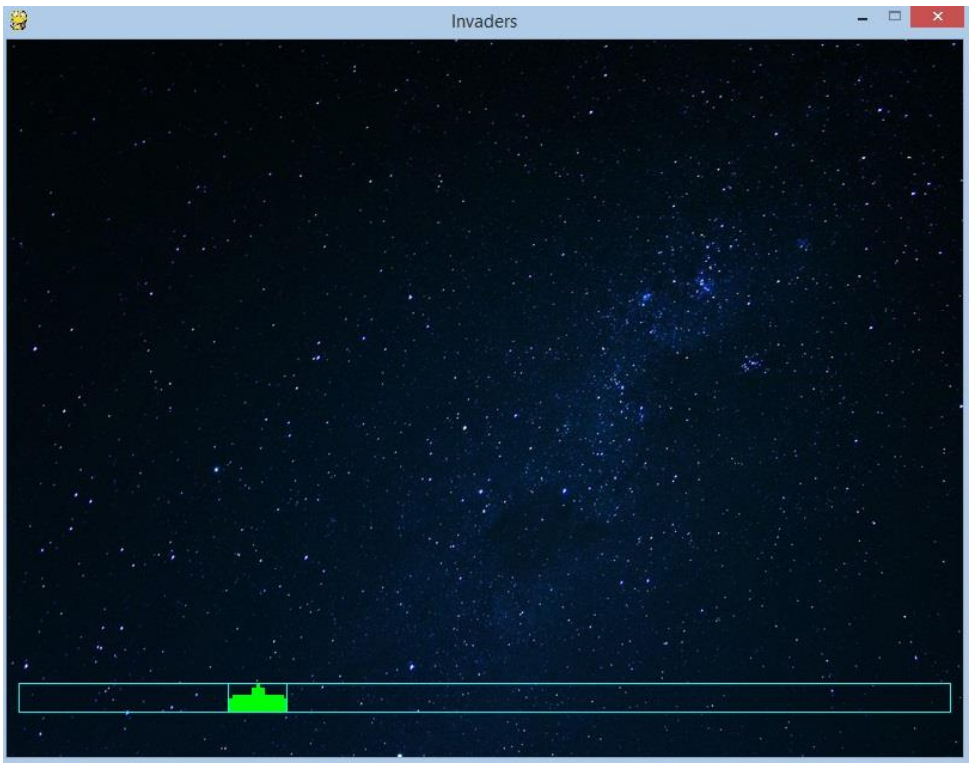
Metoda `move_ip()` pomoću pravokutnog okvira pomiče objekt u zadanom pomaku: `move_ip(x, y)`. Budući da se letjelica igrača kreće samo vodoravno, za `y` vrijednost ćemo u obje situacije dodijeliti nulu. Metoda `clamp_ip()` omogućava pomicanje letjelice samo unutar zadanog pravokutnog okvira `bounds`. Na taj način letjelica ne može izaći izvan ekrana.[31]

[31] https://www.pygame.org/docs/ref/rect.html

Za pokretanje letjelice još su potrebne dvije preinake. U metodi `display_frame()` moramo pratiti je li igrač pritisnuo tipku na tipkovnici pri svakom *frameu*. Funkcija `get_pressed()` dobiva stanje svih tipki na tipkovnici. Rezultat funkcije je n-torka s vrijednostima 0 i 1. Ako tipka nije pritisnuta vrijednost je 0, ako jest, vrijednost je 1 za svaku tipku na tipkovnici.

Budući da smo nadjačali metodu `update()` sada tu metodu i pozivamo u svakom *frameu* šaljući argument `keys` kao rezultat funkcije `get_pressed()`.

```
keys = pg.key.get_pressed()
self.all_sprites.update(keys)
```



Slika 3: Letjelica u zadanim granicama

6. IZRADA NEPRIJATELJSKE VOJSKE, *INVADERA*

6.1. Prikaz invadera

Na početku ćemo crtati samo jednog *invadera*. Klasa `Invader` bit će vrlo slična klasi `Player`. Konstruktor ima parametar `image`, a atributi su `size`, `image` i `rect`.

```
class Invader(pg.sprite.Sprite):
    def __init__(self, image):
        super().__init__()
        self.size = (26, 26)
        self.image = pg.transform.scale(image, self.size)
        self.rect = self.image.get_rect(center=INVADER_START_POS)
```

Za razliku od klase `Player`, gdje smo klasnim atributom odredili početnu poziciju letjelice, početna pozicija *invadera* bit će fiksirana, pa ćemo je definirati na početku programa kao konstantu: `INVADER_START_POS = (170, 100)`.

U konstruktoru klase `Game` kreirat ćemo posebnu grupu `invader_grp` i u nju dodati instancu klase `Invader` u metodi `start_game()`. Zatim ćemo novu grupu dodati grupi `all_sprites` kojom ažuriramo i crtamo sve *spriteove*. Grupa `all_sprites` se puni korištenjem naslijeđene metode `add()` klase `Group`.[32]

```
def start_game(self):
    ...
    self.invader_grp.add(Invader(self.images["a0"]))
    self.all_sprites.add(self.invader_grp)
```

6.2. Prikaz grupe invadera

Umjesto jednog, inicijalizirat ćemo više *invadera*. Najprije ćemo fiksirati broj redaka i stupaca s *invaderima* kao i prazni prostor između njih.

```
INVADER_ROWS, INVADER_COLUMNS = (5, 11)
INVADER_GAP = (46, 41)
```

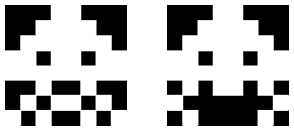
U konstruktoru klase `Invader` dodat ćemo parametar `position` kako bi svaki pojedini *invader* imao jedinstveni položaj.

[32] https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.add

Pomoću metode `create_invaders()` kreirat ćemo *invadere* koristeći dvostruku `for` petlju. Prema broju redaka, stupaca i udaljenosti između *invadera*, određuje se položaj za svakog pojedinog *invadera*, stvara se objekt te se pune grupe.

```
def create_invaders(self):
    for row in range(INVADER_ROWS):
        for column in range(INVADER_COLUMNS):
            invader_pos = (
                INVADER_START_POS[0] + column * INVADER_GAP[0],
                INVADER_START_POS[1] + row * INVADER_GAP[1])
            image = self.images["a0"]
            invader = Invader(image, invader_pos)
            self.invader_grp.add(invader)
    self.all_sprites.add(self.invader_grp)
```

6.3. Animacija invadera



Slika 4: Dvije slike istog invadera

Kako bi se postigao efekt animacije, umjesto jedne, svakog ćemo *invadera* inicijalizirati s listom od dvije slike. Da bi to postigli u `create_invaders()` metodi moramo prije kreiranja instance svakog *invadera* učitati dvije slike:

```
images = [self.images["a0"], self.images["a1"]]
invader = Invader(images, invader_pos)
```

U konstruktoru klase `Invader` dodajemo atribut `self.images` koji će biti lista dvije slike. Listu generiramo posebnom metodom koja se zove *razumijevanje liste* (eng. *list comprehension*). Postavljamo početni indeks liste slika (*0*) i pripremamo prvu sliku iz liste za prikaz.

```
self.images = [pg.transform.scale(img, self.size) for img in images]
self.index = 0
self.image = self.images[self.index]
```

Animaciju ćemo izvesti pomoću metode `animate()`. Animacija je u stvari samo neprekidno izmjenjivanje dviju slika. Izmjenjujemo indekse na listi slika naizmjence: 0 -> 1 -> 0 -> 1 -> 0 ->... Na taj se način prilikom svakog poziva metodi, izmijeni i slika koju treba prikazati.

```
    def animate(self):
        self.index = 1 - self.index
        self.image = self.images[self.index]
```

Nadjačanoj metodi `update()` dodajemo parametar `time_to_move` koji je *Boolean* i određuje treba li pomicati *invadere*.

```
    def update(self, keys, time_to_move, *args):
        if time_to_move:
            self.animate()
```

Kako bismo pokrenuli animaciju *invadera*, u klasi `Game` dodajemo dva nova atributa. Atribut `invaders_animate_time` će pratiti koliko treba proći vremena između dva pomicanja *invadera* (u milisekundama). U ovom trenutku *invaderi* će se pomicati svakih 800 milisekundi.

```
    self.invaders_animate_time = 800
```

Atribut `invaders_last_move_time` će na početku zapamtiti točan trenutak kad je kreirana grupa. Korištenjem funkcije `get_ticks()` dobit ćemo točno vrijeme (u milisekundama) od trenutka inicijaliziranja potrebnih Pygame modula (`pg.init()`) do trenutka poziva funkcije (na našem računalu u pravilu između 220 i 250 milisekundi).[33]

```
    self.invaders_last_move_time = pg.time.get_ticks()
```

Prilikom svakog *framea* atribut `time_to_move_invaders` provjeravat će je li prošlo dovoljno vremena za sljedeće ažuriranje *invadera*. Ako jest, atribut `invaders_last_move_time` treba ažurirati.

```
    time_to_move_invaders = current_time - self.invaders_last_move_time >
                                        self.invaders_animate_time

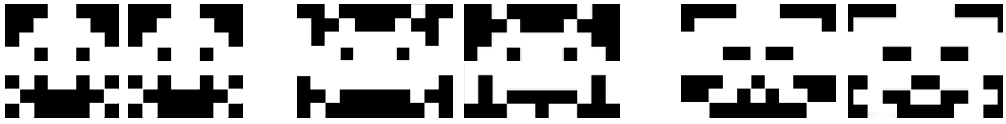
    if time_to_move_invaders:
        self.process_invaders()
```

Metoda `process_invaders()` ažurira atribut `invaders_last_move_time`, te ga povećava za novih 800 milisekundi.

```
def process_invaders(self):
    self.invaders_last_move_time += self.invaders_animate_time
```

[33] https://www.pygame.org/docs/ref/time.html

6.4. Prikaz grupe različitih invadera



Invader 'a' *Invader 'b'* *Invader 'c'*

Slika 5: Svi invaderi u igri

Dosad smo učitavali i animirali samo prvu vrstu *invadera*. No sad ćemo učitati i ostale dvije vrste te ih smjestiti u pet pripadajućih redova. Konfiguraciju ćemo definirati u rječniku `INVADERS_CONFIG`. Slagat ćemo *invadere* po redovima:

{0. red: (("Invader 'a' slika 0", "Invader 'a' slika 1"), (širina, visina)),

1. red: (("Invader 'b' slika 0", "Invader 'b' slika 1"), (širina, visina)),

2. red: ...

... }

Key rječnika je *integer* koji predstavlja red u kojem se crta *invader*. *Value* rječnika je ugniježđena n-torka, koja sadrži druge dvije n-torke: prva n-torka se sastoji od dva *stringa*, to su nazivi slika za *invadera* u tom redu, a druga n-torka je veličina slike u pikselima.

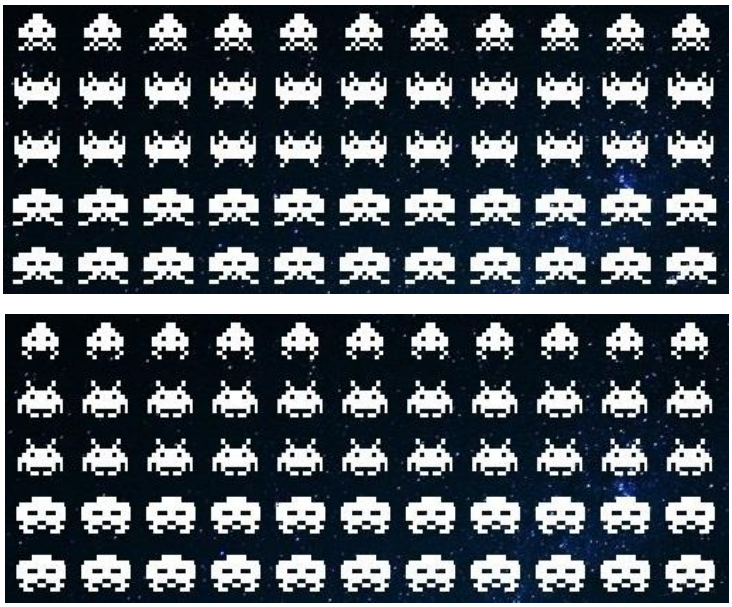
```
INVADERS_CONFIG = {
    0: (("a0", "a1"), (26, 26)),
    1: (("b0", "b1"), (32, 26)),
    2: (("b0", "b1"), (32, 26)),
    3: (("c0", "c1"), (35, 26)),
    4: (("c0", "c1"), (35, 26))
}
```

U klasi `Invader` mičemo atribut instance `size`, jer neće svi *invaderi* imati istu veličinu, te dodajemo parametar `size` u konstruktor.

```
class Invader(pg.sprite.Sprite):
    def __init__(self, images, position, size):
    ...
```

U klasi `Game` u metodi `create_invaders()` umjesto učitavanja slika "a0" i "a1", svaki će *invader* sada dobiti svoje slike iz `INVADER_CONFIG` rječnika, zavisno od reda gdje se nalazi. Iz rječnika ćemo izvaditi i veličinu za svakog *invadera*, te ju koristiti za inicijalizaciju.

```
images = [self.images[img] for img in INVADERS_CONFIG[row][0]]
size = INVADERS_CONFIG[row][1]
invader = Invader(images, invader_pos, size)
```



Slika 6: Animirani invaderi konfigurirani po retcima i stupcima

6.5. Vodoravno pomicanje invadera

U klasi `Invader` dodat ćemo klasni atribut `move_x = 10`, te u `update()` metodi pomicati *invadere* za 10 piksela udesno, koristeći već od ranije poznatu `move_ip()` metodu.

```
self.rect.move_ip(self.move_x, 0)
```

Međutim, kada *invaderi* dođu do ruba prozora, oni se neće okrenuti, već će nastaviti pokret izvan prozora. Kako bismo to preduhitrili morat ćemo napraviti nekoliko promjena. Prvo dodajemo lijevu i desnu granicu do koje smiju ići *invaderi*.

```
self.left_bound, self.right_bound = (22, 778)
```

Zatim ćemo dodati metodu `is_on_edge()` koja ispituje nalazi li se *invader* na lijevom ili desnom rubu prozora igre.

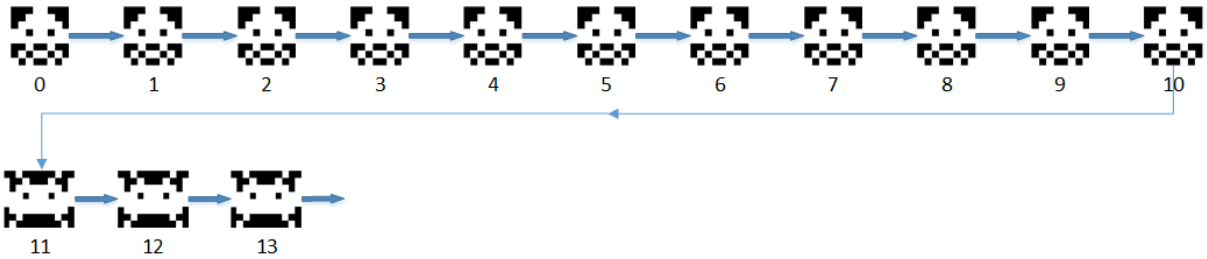
```
def is_on_edge(self):
    return self.rect.left <= self.left_bound or
           self.rect.right >= self.right_bound
```

Kako bi znali jesu li *invaderi* na rubu prozora, trebalo bi svakog pojedinačnog ispitati je li na rubu. Međutim, to bi bilo dosta neučinkovito, jer na rubu mogu biti samo *invaderi* na graničnim stupcima, dok većina „unutrašnjih“ ne može. Iz tog razloga

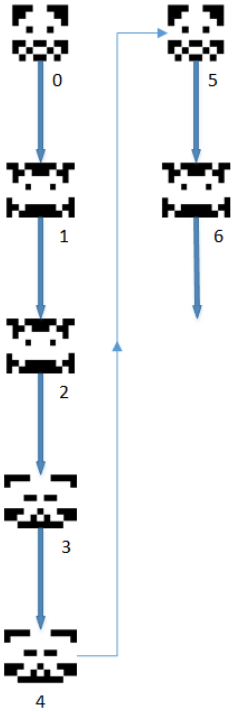
odredit ćemo samo dva granična *invadera*, ovisno o smjeru kretanja, pomoću kojih ćemo doznati je li grupa došla na rub prozora ili nije.

U metodi `create_invaders()` klase Game izmijenit ćemo dosadašnji način punjenja grupe *invadera* tako da prvo punimo po stupcima. Na taj način ćemo lakše manipulirati s rubnim *invaderima*.

```
for column in range(INVADER_COLUMNS):
    for row in range(INVADER_ROWS):
        invader_pos = (INVADER_START_POS[0] + column * INVADER_GAP[0],
                       INVADER_START_POS[1] + row * INVADER_GAP[1])
```



Slika 7: Punjenje grupe invadera po retcima



Slika 8: Punjenje grupe invadera po stupcima

Graničnog *invadera* odredit ćemo pomoću `sprites()` metode klase `Group` koja objekte grupe vraća u obliku liste.[34] Granični je onaj koji je prvi (indeks 0) ili zadnji

[34] https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.sprites

(indeks -1) u grupi, ovisno o tome u kojem se smjeru kreće grupa. Indeks određujemo ternarnim uvjetnim operatorom (eng. *ternary operator*): `a if condition else b`.

```
index = -1 if Invader.move_x > 0 else 0
boundary_invader = self.invader_grp.sprites()[index]
```

Naposljetku, ispitujemo je li granični *invader* na rubu prozora. Ako je, mijenjamo atribut `move_x` klase `Invader`, tako da *invaderi* ne izlaze iz prozora, nego se počnu kretati u suprotnom smjeru.

```
if boundary_invader.is_on_edge():
    Invader.move_x *= -1
```

6.6. Silazno pomicanje invadera

Invaderi se sada kreću naizmjence od lijevog do desnog ruba prozora. Želimo dalje unaprijediti njihovo kretanje tako da kad dođu na rub pomaknemo cijelu grupu *invadera* jedan red silazno.

Klasi `Invader` dodat ćemo klasne atribute `move`, `move_y` i `moved_down`.

```
class Invader(pg.sprite.Sprite):
    # Dodani atributi:
    # move: n-torka, sadrži move_x i move_y
    # move_y: pomicanje invadera prema dolje
    # moved_down: Boolean, jesu li invaderi pomaknuti silazno
    move = move_x, move_y = (10, 0)
    moved_down = False
    ...
```

Kada *invaderi* dođu na rub prozora, umjesto okretanja smjera pomicanja, trebaju otići okomito jedan red dolje. Tek nakon toga okrećemo smjer pomicanja. Sve to postižemo manipulacijom atributa klase `Invader` iz metode `process_invaders()`.

```
if boundary_invader.is_on_edge() and not Invader.moved_down:
    Invader.move = (0, 30)
    Invader.moved_down = True
elif Invader.moved_down:
    Invader.move_x *= -1
    Invader.move = (Invader.move_x, Invader.move_y)
    Invader.moved_down = False
```

Trenutno *invaderi* „ispadaju“ izvan prozora kada se spuste. Kontrolu toga ćemo namjestiti naknadno.

7. RAKETE I PUCANJE

7.1. Pucanje letjelice igrača

Rakete ćemo „proizvoditi“ u klasi `Missile`. Raketa neće biti kao do sada ranije pripremljena slika, već mali pravokutnik ispunjen bijelom bojom, veličine 3x10 piksela.

Klasa će uz konstruktor imati još dvije metode. Metoda `is_off_screen()` provjerava je li raketa dodirnula vrh prozora, te vraća Boolean. Nadjačana metoda `update()` ne traži nikakve dodatne parametre. Pomiče raketu okomito po zadanoj vrijednosti atributa `move_y`, te poziva metodu `is_off_screen()`. Ako je raketa dodirnula vrh, onda ju uklanja iz svih grupa.

```
class Missile(pg.sprite.Sprite):
    def __init__(self, position, move):
        super().__init__()
        self.size = (3, 10)
        self.image = pg.Surface(self.size)
        self.image.fill(pg.Color("White"))
        self.rect = self.image.get_rect(center=position)
        self.move_y = move

    def is_off_screen(self):
        return self.rect.top <= 0

    def update(self, *args):
        self.rect.move_ip(0, self.move_y)
        if self.is_off_screen():
            self.kill()
```

U klasi Player dodajemo atribut instance `missile_move`, koji nam je potreban pri kreiranju raketa. Budući da se raketa kreće od igrača uzlazno, dakle po y osi u negativnom smjeru, imat će negativnu vrijednost.

```
self.missile_move = -10
```

Potrebno je dodati i metodu `shoot()` koja kreira i vraća instancu klase `Missile` s argumentima položaja igrača i brzine (smjera) kretanja rakete.

```
def shoot(self):
    return Missile(self.rect.midtop, self.missile_move)
```

U klasi `Game` napisat ćemo metodu `create_player_missile()` koja poziva metodu `shoot()` klase `Player` i stvara raketu. Raketu spremamo u kontejner `player_missile_grp`, pomoću kojeg ćemo kasnije lako otkriti je li došlo do kolizije s *invaderom*.

```
def create_player_missile(self):
    missile = self.player.shoot()
    self.player_missile_grp.add(missile)
    self.all_sprites.add(missile)
```

Na kraju nam je potreban okidač za rakete. To će biti tipka *SPACE* na tipkovnici u metodi `process_events()`. Provjeravanjem je li grupa `player_missile_grp` prazna onemogućavamo višestruko pucanje.

```
if event.type == KEYDOWN and event.key == K_SPACE:
    if not self.player_missile_grp:
        self.create_player_missile()
```

7.2. Kolizije rakete i invadera

Pritiskom na tipku SPACE letjelica sada ispaljuje rakete, ali one samo prelaze preko *invadera*. To želimo promijeniti tako da kada raketa dodirne *invadera*, nastane eksplozija i pogođeni *invader* nestane. U klasi `Game` napisat ćemo metodu `check_collisions()` zaduženu za provjeru kolizija između grupa objekata u igri. Kolizije između dviju grupa objekata detektiramo funkcijom `groupcollide()`. Funkcija prima pet argumenata, od kojih su četiri obvezni. Prva dva argumenta su grupe *spriteova* između kojih provjeravamo kolizije. Treći i četvrti argument su Booleani, s kojima utvrđujemo da li *spriteove* uklanjamo ili ne. Funkcija vraća rječnik čiji ključevi su svi *spriteovi* prve grupe koji su u koliziji, dok je vrijednost rječnika lista *spriteova* druge grupe koji su u koliziji sa *spriteom* prve grupe.[35] Budući da dozvoljavamo ispucavanje samo jedne rakete u vremenu, u listi može biti samo jedan *invader*. Ako nema kolizije funkcija vraća prazan rječnik.

```
missile_invader_collision = pg.sprite.groupcollide(
                                    self.player_missile_grp,
                                    self.invader_grp,
                                    True, False)
```

Kada dođe do kolizije između rakete i *invadera*, raketa se briše, a *invadera* ćemo obraditi, pa potom obrisati (`True`, `False`). Ako se *invader* nalazi u listi, koristit ćemo *list slicing* (isijecanje liste) kako bismo doprli do njega. Vrijednost rječnika nije standardna lista, nego *view objekt*, stoga ga moramo prvo pretvoriti u listu prije *slicinga*.

[35] https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.groupcollide

```
    collided_invader = list(missile_invader_collision.values())[0][0]
    self.process_missile_invader_collision(collided_invader)
```

Invader u koliziji postat će argument pri pozivu nove metode `process_missile_invader_collision()`. U njoj uzimamo poziciju pogođenog *invadera*, koja nam je potrebna za kreiranje eksplozije. Nakon uklanjanja pogođenog *invadera* kreiramo instancu klase `Explosion`, te ju dodajemo u grupu `all_sprites`. Ako nema više *invadera* na zaslonu, tj. svi su „upucani“, pozivamo metodu `reset_invaders()`.

```
    def process_missile_invader_collision(self, collided_invader):
        collided_invader_pos = collided_invader.rect.center
        collided_invader.kill()
        self.all_sprites.add(Explosion(self.images["explosion"],
                                       collided_invader_pos))
        if not self.invader_grp:
            self.reset_invaders()
```

Pomoću klase `Explosion` kreirat ćemo eksplozije, tj. prikazivati sliku eksplozije. Eksplozija će biti vidljiva kratko vrijeme na ekranu (200 miliskeundi) u trenutku kad je pogođen *invader*. Uz uobičajene atribute, u konstruktoru ćemo definirati i atribut trajanja eksplozije, kao i atribut koji pamti točan trenutak kad je eksplozija nastala. Nadjačanoj metodi `update()` dodat ćemo parametar `current_time` kako bismo kontrolirali duljinu prikaza eksplozije.

```
    class Explosion(pg.sprite.Sprite):
        def __init__(self, image, position):
            super().__init__()
            self.size = (26, 26)
            self.image = pg.transform.scale(image, self.size)
            self.rect = self.image.get_rect(center=position)
            self.duration = 200
            self.time_created = pg.time.get_ticks()

        def update(self, keys, time_to_move, current_time, *args):
            if current_time - self.time_created > self.duration:
                self.kill()
```

Kada su svi *invaderi* pogođeni želimo nastaviti igru i ponovno nacrtati početnu grupu *invadera* na početnoj poziciji. Zato ćemo u metodi `reset_invaders()` klase `Game` „resetirati“ atribute klase `Invader` i pozvati metodu `create_invaders()`.

```
    def reset_invaders(self):
        Invader.move = Invader.move_x, Invader.move_y = (10, 0)
        Invader.moved_down = False
        self.create_invaders()
```

U metodi display_frame() pozivamo metodu check_collisions() kako bi se kolizije stalno, u svakom *frameu* provjeravale.

7.3. Pucanje invadera

Pucanje *invadera* ćemo izvesti na ponešto drugačiji način od pucanja letjelice igrača. Ono će zavisiti o generiranom slučajnom broju unutar svakog *framea*, tj. možemo reći da ćemo pucanje *invadera* poslučajiti.

Modul `random`, kako smo već na početku spomenuli, generira pseudoslučajne brojeve. Za naše potrebe uključit ćemo dvije funkcije. Funkcija `randint`, vraća slučajan cijeli broj *N* u rasponu brojeva *a* i *b* (*a* <= *N* <= *b*), dok funkcija `choice` vraća slučajni element neke sekvence.[36]

```
>>> from random import randint, choice
>>> randint(5, 150)
71
>>> choice("Python")
't'
>>> choice([1, 2, 3, 4, 5])
2
>>>
```

Kao i kod letjelice igrača, u klasi `Invader` dodajemo atribut `missile_move` koji određuje brzinu padanja rakete, te metodu `shoot()` koja kreira instancu klase `Missile`. Zatim ćemo u klasi `Game` napisati metodu `create_invader_missile()` pomoću koje ćemo *invaderima* omogućiti ispaljivanje raketa. Metoda će se pozivati svaki *frame*. Na početku se izabire slučajan broj u rasponu od 1 do 1000. Ako je taj broj manji od 10, slučajnim odabirom izabire se *invader* koji poziva metodu `shoot()` klase `Invader`. Kreira se objekt rakete i sprema u grupe.

```
def create_invader_missile(self):
    num = randint(1, 1000)

    if num < 10:
        invader_to_shoot = choice(self.invader_grp.sprites())
        invader_missile = invader_to_shoot.shoot()
        self.invader_missile_grp.add(invader_missile)
        self.all_sprites.add(invader_missile)
```

[36] https://docs.python.org/3/library/random.html

U metodi `is_off_screen()` klase `Missile` mijenjamo uvjet kako bi rakete koje su ispalili *invaderi*, a koje su došle do dna prozora bile obrisane.

```
    def is_off_screen(self):
        return self.rect.top <= 0 or self.rect.bottom >= 600
```

7.4. Pucanje samo najnižih invadera po stupcima

U ovom trenutku pucaju svi *invaderi*. Želimo postići da pucaju samo *invaderi* koji su najniži po stupcima, tj. oni koji su najbliže letjelici igrača.

U klasi Game napravit ćemo posebnu grupu `lowermost_invaders`, u kojoj će se nalaziti samo najniži *invaderi* po stupcima. Ta se grupa puni odmah prilikom kreiranja *invadera* u metodi `create_invaders()`.

```
    if row == INVADER_ROWS - 1:
        self.lowermost_invaders.add(invader)
```

Slučajni odabir *invadera* za pucanje u metodi `create_invader_missile()` sad je samo iz grupe najnižih.

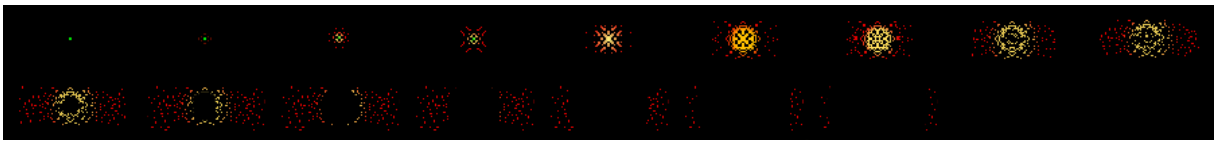
```
    if num < 10:
        invader_to_shoot = choice(self.lowermost_invaders.sprites())
        ...
```

Ako je pogođen *invader* iz grupe `lowermost_invaders`, njega treba zamijeniti *invaderom* iznad njega, tj. prvim sljedećim najnižim *invaderom* na stupcu na kojem se pogodak desio. Zbog toga u metodi `process_missile_invader_collision()` uzimamo indeks pogođenog *invadera* iz grupe svih *invadera*. Dodajemo prvog sljedećeg najnižeg *invadera* u istom stupcu, `next_lowermost_invader`, prema indeksu grupe. Novi *invader* ulazi u *lowermost* grupu, samo ako nije već cijeli stupac *invadera* eliminiran, tj. nisu svi drugi *invaderi* u stupcu već ranije pogođeni.

```
def process_missile_invader_collision(self, collided_invader):
    if collided_invader in self.lowermost_invaders:
        collided_invader_idx =
                    self.invader_grp.sprites().index(collided_invader)
        next_lowermost_invader =
                    self.invader_grp.sprites()[collided_invader_idx - 1]
        if next_lowermost_invader not in self.lowermost_invaders:
            self.lowermost_invaders.add(next_lowermost_invader)
```

7.5. Kolizije rakete i letjelice, spritesheet

Prilikom kolizije rakete invadera i letjelice igrača želimo postići efekt animirane eksplozije. To ćemo postići koristeći *spritesheet*, tj. veće slike, koja sadržava nekoliko manjih sličica iste veličine za kreiranje animacija.



Slika 9: Spritesheet za eksploziju letjelice

Umjesto da se animacija standardno napravi kroz učitavanje i obradu više pojedinačnih manjih slika, zamisao *spritesheeta* je da se animacija postigne učitavanjem i obradom samo jedne, ali veće slike.

U konstruktoru nove klase `SpriteSheet` stavili smo nekoliko parametara s *defaultnim* vrijednostima: `size=(1350, 150)`, `sprites_in_row=9`, `rows=2`. *Defaultne* vrijednosti parametara omogućuju da se odgovarajući argumenti pri pozivanju metode mogu izostaviti. Kasnije ćemo pri kreiranju objekta zato staviti samo argumente za `image` i `position`, jer su naši parametri prilagođeni našem *spritesheetu*, koji ima devet sličica u retku i dva reda.

```
class SpriteSheet(pg.sprite.Sprite):
    def __init__(self, image, position, size=(1350, 150),
                                        sprites_in_row=9, rows=2):
        ...
```

Ako bi htjeli učitati i obraditi neki drugi *spritesheet*, drugačije veličine i s drugačijim rasporedom sličica i redova, jednostavno bismo dodali argumente za `size`, `sprites_in_row` i `row`, koji vrijede za taj novi *spritesheet*.

U metodi `create_sprites_list()` dvostrukom `for` petljom raščlanjujemo pojedine sličica *spritesheeta*. Pomoću metode `subsurface()` označavamo manje pravokutne dijelove, tj. manje sličice iz učitane velike slike kojima punimo povratnu listu.

```
def create_sprites_list(self):
    sprite_list = []
    for row in range(self.rows):
        for sprite in range(self.sprites_in_row):
            sprite_list.append(self.sprites_sheet.subsurface(
                 self.sprite_width * sprite, self.sprite_height * row,
                 self.sprite_width, self.sprite_height))
    return sprite_list
```

Brzina izmjenjivanja pojedinih sličica je 80 milisekundi, a cijela animacija završava nakon 1500 milisekundi. Kada prođe potrebno vrijeme od početka eksplozije, uklanjamo objekt eksplozije iz svih grupa.

```
def update(self, keys, time_to_move, current_time, *args):
    if current_time - self.time_created < 1500:
        if current_time - self.last_sprite_time > 80:
            # Budući da je 18 sličica u spritesheetu,
            # pomičemo indeks operatorom modulo.
            self.index = self.counter % 18
            self.image = self.sprite_list[self.index]
            self.counter += 1
            self.last_sprite_time += 80
    else:
        self.kill()
```

U klasi `Game` dodajemo nekoliko novih atributa. Zastavica `player_destroyed` promijenit će stanje u True, kad je igrač pogođen. Atribut `player_destroyed_time` će zabilježiti trenutak kad je letjelica igrača uništena, a atribut `player_reset_time` određuje vrijeme od trenutka kad je igrač pogođen do njegova *resetiranja*.

U metodi `check_collisions()` detektiramo koliziju između rakete *invadera* i letjelice igrača.

```
missile_player_collision = pg.sprite.groupcollide(
                                            self.invader_missile_grp,
                                            self.player_grp,
                                            True, False)
```

Ako se susretnu raketa igrača i raketa *invadera*, obje ćemo obrisati.

```
pg.sprite.groupcollide(self.player_missile_grp, self.invader_missile_grp,
                       True, True)
```

Ako je došlo do kolizije između rakete *invadera* i igrača, ne trebamo izvlačiti objekt *playera*, kao sa `collided_invader`, budući da je *player* već atribut instance klase `Game`: `self.player`.

```
    if missile_player_collision:
        self.process_missile_player_collision()
```

U metodi `process_missile_player_collision()` bilježimo trenutak uništavanja igrača, aktiviramo zastavicu `player_destroyed`, te uzimamo poziciju pogođene letjelice, koja nam je potrebna za kreiranje eksplozije.

```
    def process_missile_player_collision(self):
        self.player_destroyed_time = pg.time.get_ticks()
        self.player_destroyed = True
        player_pos = self.player.rect.center
        Player.position = player_pos
        self.player.kill()
        self.all_sprites.add(SpriteSheet(self.images["explosion_9x2"],
                                         player_pos))
```

Metoda `reset_player()` će *resetirati* letjelicu igrača, tako da se nova letjelica kreira dvije sekunde nakon što je prethodna pogođena.

```
def reset_player(self, current_time):
    time_to_reset_player = current_time - self.player_destroyed_time >
                               self.player_reset_time

    if time_to_reset_player:
        self.create_player()
        self.player_destroyed = False
```

Je li došao trenutak za *resetiranje* igrača provjeravamo u svakom *frameu*.

```
    if self.player_destroyed:
        self.reset_player(current_time)
```

8. IZRADA BONUS LETJELICE, MYSTERY

8.1. Prikaz mystery letjelice



Slika 10: Mystery letjelica

Mystery je posebna letjelica koja se povremeno pojavljuje na vrhu ekrana. Ne ispaljuje rakete i tako ne predstavlja opasnost za igrača. Kad ju igrač uspije pogoditi, nagrađen je s bonus bodovima u rasponu od 50 do 300 bodova.

Najprije ćemo nacrtati statičnu *mystery* letjelicu na sredini prozora. U konstruktoru klase `Mystery` na već uobičajen način definiramo atribute instance `size`, `image` i `rect`, te klasni atribut `pos` za poziciju.

```
class Mystery(pg.sprite.Sprite):
    pos = (400, 55)

    def __init__(self, image):
        super().__init__()
        self.size = (50, 20)
        self.image = pg.transform.scale(image, self.size).convert()
        self.rect = self.image.get_rect(center=self.pos)
```

U klasi `Game` u `start_game()` metodi kreiramo objekt *mystery* letjelice i dodajemo ga u grupu `all_sprites`.

```
self.mystery = Mystery(self.images["mystery"])
self.all_sprites.add(self.mystery)
```

8.2. Pomicanje mystery letjelice s lijeva na desno

Prije nego počnemo pomicati *mystery* letjelicu moramo je smjestiti tik uz desni gornji rub izvan prozora. Također joj moramo odrediti brzinu, odnosno smjer kretanja.

```
class Mystery(pg.sprite.Sprite):
    pos = (-25, 55)
    move_x = 3
```

Metoda `off_screen()` će utvrđivati je li *mystery* u granicama prozora.

```
def off_screen(self):
    return self.rect.right <= self.left_bound or
            self.rect.left >= self.right_bound
```

U klasi `Game` dodajemo dva nova atributa: `mystery_created_time` koji će pamtiti točno vrijeme stvaranja *mysterija* i `mystery_animate_time` koji određuje pojavljivanje *mysterija* slučajnim odabirom vremena u rasponu između 7,5 i 20 sekundi. Umjesto u `start_game()` metodi, *mystery* ćemo instancirati u `create_mystery()` metodi.

```
def create_mystery(self):
    self.mystery = Mystery(self.images["mystery"])
    self.mystery_grp.add(self.mystery)
    self.all_sprites.add(self.mystery)
    self.mystery_animate_time = randint(7500, 20000)
    self.mystery_created_time = pg.time.get_ticks()
```

Isto kao i s *invaderima*, provjeravamo je li došlo vrijeme za pojavljivanje *mysterija* u `display_frame()` metodi, te ako je, pozivamo `create_mystery()` metodu.

```
time_to_create_mystery = current_time - self.mystery_created_time >
                         self.mystery_animate_time
if time_to_create_mystery:
    self.create_mystery()
```

8.3. Pomicanje mystery letjelice u oba smjera i kolizija s raketom

Umjesto jedne predefinirane pozicije, postavit ćemo dvije, ovisno o tome želimo li da *mystery* krene s lijeve ili desne strane.

```
class Mystery(pg.sprite.Sprite):
    pos = pos_left = (-25, 55)
    pos_right = (825, 55)
```

Kako bismo igraču malo otežali pogotke *mystery* letjelice, pratit ćemo smjer kretanja *invadera*, te ovisno o njemu odrediti i smjer kretanja *mystery* letjelice.

```
def create_mystery(self):
    if Invader.move_x > 0:
        Mystery.pos = Mystery.pos_left
        Mystery.move_x = 3
    else:
        Mystery.pos = Mystery.pos_right
        Mystery.move_x = -3
```

Dodajemo prepoznavanje kolizije između rakete igrača i *mysterija*. Zasad brišemo oba objekta, a kasnije ćemo implementirati bodovanje i animaciju.

```
missile_mystery_collision = pg.sprite.groupcollide(
                                        self.player_missile_grp,
                                        self.mystery_grp,
                                        True, True)
```

9. TEKST, REZULTAT I ŽIVOTI IGRAČA, KRAJ IGRE

9.1. Prikaz statičnog teksta

U ovom trenutku igra je već prilično funkcionalna; *spriteovi* se neovisno jedni o drugima kreću po prozoru, te su riješene kolizije. Stoga sad želimo malo uljepšati izgled igre prikazom teksta.

Klasa `Text` bit će osnovna klasa za prikaz teksta u prozoru igre. Konstruktor klase definirat će tekst i njegovu poziciju, font teksta, te veličinu i boju fonta.[37] Pomoću `render()` metode ćemo pretvoriti tekst u sliku. Budući da će tekst biti statičan, nije potrebno pisati `update()` metodu.

```
class Text(pg.sprite.Sprite):
    def __init__(self, txt, pos, font, font_size=16, font_color="Cyan"):
        super().__init__()
        self.pos = pos
        self.font = pg.font.Font(font, font_size)
        self.font_color = pg.Color(font_color)
        self.image = self.font.render(txt, True, self.font_color)
        self.rect = self.image.get_rect(topleft=self.pos)
```

U metodi `create_text()` klase `Game` kreirat ćemo dvije instance klase `Text`, te ih dodati novoj grupi `text_grp`.

```
     def create_text(self):
         score_text = Text("SCORE", (15, 15), FONT)
         lives_text = Text("LIVES", (730, 15), FONT)
         self.text_grp.add(score_text, lives_text)
         self.all_sprites.add(score_text, lives_text)
```

9.2. Prikaz rezultata, dinamički tekst

Budući da želimo prikazivati uspjeh igrača, tj. njegov trenutni rezultat, napravit ćemo novu klasu `UpdatedText` koja će nasljeđivati prethodno kreiranu klasu `Text`. U novoj podklasi neće biti novih atributa, pa stoga nećemo niti pisati konstruktor. Metoda `update()` imat će novi parametar `txt`, u obliku *stringa*. To će biti trenutni rezultat igrača koji ćemo slati u svakom *frameu*, a koji će se onda renderirati i na taj način prikazivati trenutačne bodove igrača.

[37] Font koji koristimo u igrici preuzet je s https://fonts2u.com/space-invaders-regular.font

```
class UpdatedText(Text):
    def update(self, keys, time_to_move, current_time, txt, *args):
        self.image = self.font.render(txt, True, self.font_color)
```

Kako bi znali koliko donosi svaki pogodak, u konfiguracijskom rječniku *invadera* dodat ćemo bodove. Najviše bodova donose *invaderi* koje je najteže pogoditi, tj. oni u najgornjem, nultom redu.

```
# red: ((slike), (veličina), bodovi)
INVADERS_CONFIG = {
    0: (("a0", "a1"), (26, 26), 40),
    1: (("b0", "b1"), (32, 26), 20),
    2: (("b0", "b1"), (32, 26), 20),
    3: (("c0", "c1"), (35, 26), 10),
    4: (("c0", "c1"), (35, 26), 10)
}
```

U klasi `Invader` dodajemo parametar `row` kako bismo prilikom kolizije znali koliko bodova treba dodati za pogođenog *invadera*.

U klasi `Game` dodajemo novi numerički atribut za praćenje bodova: `player_score`. U prethodno kreiranoj metodi `create_text()` dodajemo instancu podklase `UpdatedText`. Tekst koji se u klasi renderira mora biti *string*, a budući da je atribut `player_score` *integer*, koristit ćemo *f-string* za prikaz rezultata..

```
player_score_text = UpdatedText(f"{self.player_score}", (80, 15), FONT)
```

Potom ćemo kreirati dvije jednostavne metode za obradu rezultata. Metoda `calculate_score()` vraća broj bodova za pogođenog *invadera* s obzirom na red u kojem je bio, uzimajući bodove iz konstante INVADERS_CONFIG. Metoda `update_score()` ažurira atribut instance `player_score`, tj. trenutni rezultat igrača.

```
def calculate_score(self, row):
    return INVADERS_CONFIG[row][2]

def update_score(self, score):
    self.player_score += score
```

Ažuriranje rezultata dešava se u trenutku kad je igrač pogodio *invadera*. To znači da ćemo u metodi za obradu kolizija `process_missile_invader_collision()` pozivati prethodno napisane metode.

```
self.update_score(self.calculate_score(collided_invader.row))
```

9.3. Prikaz trepćućeg teksta

Još nam je preostalo za riješiti pogađanje *mystery* letjelice, te njeno bodovanje. Pri svakom pogotku *mystery* letjelice želimo da se na tom mjestu pojavi treptajući broj bodova koje je igrač osvojio. Napisat ćemo klasu `BlinkingText` koja nasljeđuje prethodno napisanu klasu `Text`. Kao i ranije, tekst se pretvara u sliku, a onda se u `update()` metodi postiže treperenje te slike. U konstruktoru kreiramo dvije slike, jednu s renderiranim brojem bodova, a drugu praznu. Ukupno trajanje kreiranog objekta bit će 1500, dok će treptanje prestajati nakon 750 milisekundi.

Treptanje teksta izvodi se u `update()` metodi na način da se svakih 50 milisekundi prikazuje slika sa brojem bodova. Koristeći operator *modulo* dobivamo ostatak dijeljenja koji uspoređujemo sa zamišljenim vremenom prikaza renderiranog teksta. Ako je ostatak dijeljenja manji od 50, prikazujemo sliku, tj. renderirani tekst, a ako je ostatak dijeljenja veći od 50, prikazujemo praznu sliku, tj. ništa (*`else`* grana).

Budući da je sat u igri namješten na 60 *frameova* u sekundi, metoda `update()` će se izvršavati svakih ~16.67 milisekundi (*1000 / 60*). U 50 milisekundi, koliko je `display_txt_image_time` atribut, metoda bi trebala tri puta prikazati sliku s tekstom (50 / (1000 / 60)). Potom će se tri *framea* zaredom prikazati prazna slika. Pa onda opet slika s tekstom, itd. Nakon 750 milisekundi, kad je zamišljeno da treptanje prestane (`stop_blinking_time`), metoda će do kraja izvođenja (`total_time`) prikazivati samo renderirani tekst, bez treptanja.

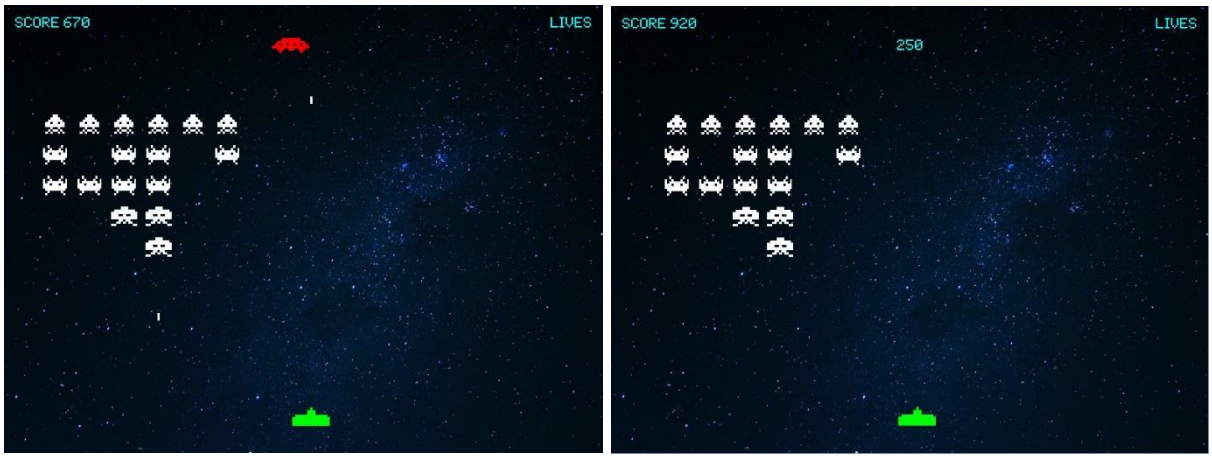
```
def update(self, keys, time_to_move, current_time, *args):
    if current_time - self.time_created > self.total_time:
        self.kill()
    elif current_time % 100 < self.display_txt_image_time or \
           current_time - self.time_created > self.stop_blinking_time:
        self.image = self.txt_image
    else:
        self.image = self.blank_image
```

Broj bodova koje igrač dobije za pogođenu *mystery* letjelicu računa se na drugačiji način od *invadera*. Stoga moramo izmijeniti `calculate_score()` metodu klase `Game`. Ako umjesto broja retka metoda primi *string* 'Mystery', izabere se i vraća slučajan broj iz liste brojeva.

```
    def calculate_score(self, row):
        if row == "Mystery":
            return choice([50, 100, 150, 200, 250, 300])
        else:
            return INVADERS_CONFIG[row][2]
```

Moramo dodati i novu metodu za ispitivanje kolizija rakete igrača i *mystery* letjelice `process_missile_mystery_collision()` u kojoj ćemo kreirati objekt `BlinkingText` klase.

```
def process_missile_mystery_collision(self):
    mystery_pos = self.mystery.rect.center
    self.mystery.kill()
    score = self.calculate_score("Mystery")
    self.update_score(score)
    self.all_sprites.add(BlinkingText(f"{score}", mystery_pos, FONT))
```



Slika 11: Pogodak mystery letjelice

9.4. Prikaz života igrača

Kako bismo prikazali živote igrača u gornjem desnom kutu prozora igre napravili smo jednostavnu klasu `PlayerLife` sa samo dva atributa: `image` i `rect`.

Odredili smo da igrač ima tri života. Jedan koji mu se dodijeli na početku igre, te još dva dodatna koja želimo prikazati. Zato ćemo u konstruktoru klase `Game` definirati `self.player_remaining_lives = 2` te dodati grupu `life_grp`. Pomoću metode `create_lives()`, crtat ćemo preostale živote igrača. Nakon što predefiniramo

početnu poziciju, te udaljenost između slika života, u petlji crtamo živote s desna na lijevo.

```
    def create_lives(self, lives):
        pos_x, pox_y = (705, 23)
        distance = 40

        # U petlji crtamo živote s desna na lijevo:
        # i = 0 -> (pos_x - i * distance, pox_y) -> (705, 23)
        # i = 1 -> (pos_x - i * distance, pox_y) -> (665, 23)
        for i in range(lives):
            player_life = PlayerLife(self.images["player"],
                                     (pos_x - i * distance, pox_y))
            self.life_grp.add(player_life)
            self.all_sprites.add(player_life)
```

9.5. Kraj igre (game over)

Igra završava ako se *invaderi* spuste prenisko ili ako igrač izgubi sve živote. Klasi `Invader` dodajemo klasni atribut `bottom_bound` koji predstavlja donju granicu koju *invaderi* ne smiju preći. Ako je pređu, igra završava.

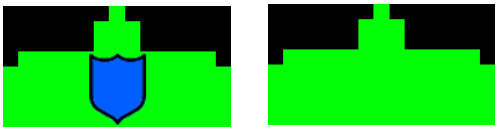
U klasi `Game` ćemo dodati zastavicu `game_over` koja će promijeniti stanje u `True`, ako neki *invader* dodirne zamišljenu donju granicu.

```
    if self.lowest_invader_position() >= Invader.bottom_bound:
        self.game_over = True
```

Metoda `lowest_invader_position()` vraća najnižeg iz grupe najnižih *invadera*.

```
    def lowest_invader_position(self):
        return max(inv.rect.bottom for inv in
                                     self.lowermost_invaders.sprites())
```

Kad je igrač pogođen, broj preostalih života treba smanjiti i moramo maknuti život iz prikaza preostalih života, ako ih ima. Zastavicom `player_shielded` ćemo aktivirati štit, koji igraču omogućava da neko vrijeme (tri sekunde) predahne, te da ne može u tom periodu opet biti pogođen.



Slika 12: Letjelica igrača sa štitom i bez štita

U metodu `reset_player()` uvodimo uvjetno grananje ovisno pod kojim uvjetima *resetiramo* igrača. Igrača nećemo *resetirati* ako mu nije preostalo više života. U tom slučaju nastupa kraj igre i mijenjamo stanje igre. Ako su prošle dvije sekunde od kad je igrač pogođen, učitavamo ga sa štitom i omogućavamo da tri sekunde ne može biti ponovno pogođen. Kad su prošle nove tri sekunde deaktiviramo štit i igra se nastavlja kao i ranije.

```
...
if time_to_reset_player:
    if self.player_remaining_lives < 0:
        self.game_over = True
    elif self.player_reset_time == 2000:
        self.create_player()
        self.player_reset_time += 3000
    else:
        Player.position = self.player.rect.center
        self.player_shielded = False
        self.player.kill()
        self.create_player()
        self.player_destroyed = False
        self.player_reset_time = 2000
```

9.6. Ponovno igranje nakon kraja igre, replay

U ovom trenutku kad nastupi kraj igre u prozoru se prikazuje samo pozadinska slika bez ikakvih obavijesti. To želimo promijeniti tako da se ispiše obavijest o kraju igre (*game over*), te da se igraču da mogućnost, ukoliko to poželi, da zaigra još jednu igru. Jednako tako želimo da tekstualni prikaz rezultata (*score*) prethodno odigrane igre ostane vidljiv.

U metodi za obradu *frameova* dodajemo uvjet koji ispituje je li nastupio kraj igre. Ako jest, želimo prikazivati samo tekstualnu grupu.

```
def display_frame(self, screen):
    ...
    if self.game_over:
        self.text_grp.draw(screen)
    else:
        ...
```

Stvorit ćemo metodu `create_game_over_screen()` za pripremu završnog prikaza. Iz `all_sprites` grupe maknut ćemo sve grupe osim tekstualne. Pripremit ćemo dva teksta za završni zaslon: „*Game Over*“ i „*Press N for new game or ESC to*

exit“. Nakon definiranja pozicije tekstova, kreirat ćemo dvije instance klase `Text` i dodat ih u tekstualnu grupu `text_grp`.

```
def create_game_over_screen(self):
    self.all_sprites.remove(self.invader_grp,
                            self.mystery_grp,
                            self.invader_missile_grp,
                            self.player_missile_grp)
    press_key_string = "Press N for new game or ESC to exit"
    press_key_pos = (SCREEN_WIDTH / 2, 450)
    press_key_text = Text(press_key_string, press_key_pos, FONT, 20)
    press_key_text.rect.center = press_key_text.pos

    game_over_string = "Game Over"
    game_over_pos = (SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2)
    game_over_text = Text(game_over_string, game_over_pos, FONT, 36)
    game_over_text.rect.center = game_over_text.pos

    self.text_grp.add(press_key_text, game_over_text)
```

Jednako tako na oba mjesta koja iniciraju kraj igre (*invaderi* su sišli do donje granice ili gubitak svih života) moramo pozivati metodu za pripremu završnog zaslona.

Ako igrač na završnom prikazu igre zaželi ponovno zaigrati, mora pritisnuti tipku „*n*“. Kad to učini u metodi za obradu događaja `process_events()` moraju se *resetirati* sve postavke, te reinicijalizirati igra.

```
def process_events(self):
    ...
    if self.game_over and event.type == KEYDOWN and event.key == K_n:
        Invader.move = Invader.move_x, Invader.move_y = (10, 0)
        Invader.moved_down = False
        Player.position = (SCREEN_WIDTH / 2, 550)
        self.__init__()
```



Slika 13: Završni prikaz igre

9.7. Nagradni život, extra life

Kako bi se igrač dodatno motivirao za igranje igre povremeno ćemo ga nagraditi s nagradnim životom. U konstruktoru klase `Game` definiramo atribut za praćenje nagradnih života: `self.extra_lives_gained = 0`. Metodu `update_score()` ćemo izmijeniti tako da igrač dobiva nagradni život nakon svakih 5000 bodova.

```
    def update_score(self, score):
        self.player_score += score

        if self.player_score // 5000 > self.extra_lives_gained:
            self.extra_life()
```

U metodi `extra_life()` povećavamo atribute sa životima, te pozivamo metodu za obradu preostalih života kako bi docrtali još jedan život.

```
    def extra_life(self):
        self.player_remaining_lives += 1
        self.extra_lives_gained += 1
        self.create_lives(1)
```

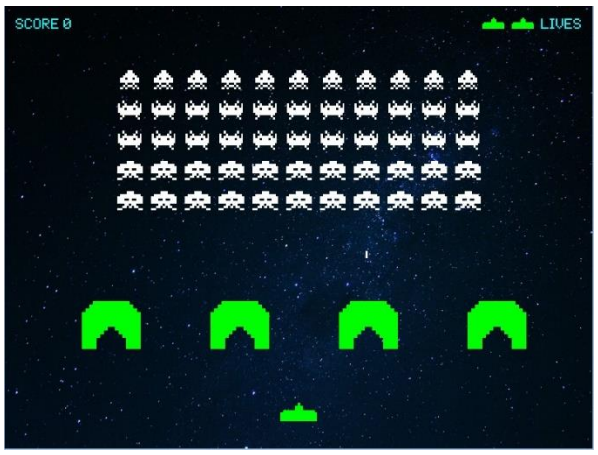
Nagradni život crtamo lijevo od posljednjeg nacrtanog života, ako ih ima.

```
    def create_lives(self, lives):
        ...
        pos_x, pox_y = (705 - distance * len(self.life_grp), 23)
        ...
```

10. BUNKERI I ZVUKOVI

10.1. Izrada bunkera

Kako bismo igraču olakšali borbu s *invaderima* postavit ćemo bunkere iznad položaja letjelice. Bunkeri će blokirati rakete koje izbacuju *invaderi* te time omogućiti igraču da se ispod njih može sakriti. Međutim, bunkeri smanjuju vidno polje igraču i slobodan prostor za pucanje, pa u tom smislu malo i otežavaju igru.



Slika 14: Početni izgled bunkera

Osnovna struktura bunkera je ugniježđena lista. Sastoji se od nula i jedinica, a svaka jedinica u listi predstavljat će mali pravokutni dio bunkera.

```
BUNKER_STRUCTURE = [
    [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
]
```

Sastavni dijelovi bunkera, bit će objekti `BunkerRect` klase. Svaki objekt će biti pravokutna sličica zelene boje veličine 5x5 piksela.

```
class BunkerRect(pg.sprite.Sprite):
    size = width, height = (5, 5)

    def __init__(self, pos):
        super().__init__()
        self.image = pg.Surface(self.size)
        self.image.fill(pg.Color("green"))
        self.rect = self.image.get_rect(topleft=pos)
```

U klasi Game napisat ćemo metodu za kreiranje bunkera. Predefinirat ćemo osnovna svojstva: broj bunkera, njihovu međusobnu udaljenost i početni položaj. Zatim ćemo pomoću trostruke `for` petlje stvarati objekte `BunkerRect` klase. U vanjskoj petlji iteriramo broj bunkera, u središnjoj petlji iteriramo redove strukture bunkera (*BUNKER_STRUCTURE*), dok u unutrašnjoj petlji iteriramo pojedine elemente svakog reda. Ako je element različit od nule, dakle ako je jedinica, kreira se instanca klase `BunkerRect` i dodaje se u grupu za bunkere.

```
def create_bunkers(self):
    total_bunkers = 4
    bunker_distance = 175
    bunker_start_pos_x, bunker_start_pos_y = (105, 400)

    for i in range(total_bunkers):
        pos_y = bunker_start_pos_y
        for row in BUNKER_STRUCTURE:
            pos_x = bunker_start_pos_x + i * bunker_distance
            for num in row:
                if num:
                    bunker_rect = BunkerRect((pos_x, pos_y))
                    self.bunker_grp.add(bunker_rect)
                pos_x += BunkerRect.width
            pos_y += BunkerRect.height
    self.all_sprites.add(self.bunker_grp)
```

Međutim, bunkeri nisu „neprobojne“ prepreke. Kada ga pogodi neka raketa jedan njegov mali dio se obriše na mjestu pogotka. Jednako tako ako se *invaderi* spuste dovoljno nisko i dodirnu bunker također će se brisati dijelovi bunkera. Zato ćemo dodati tri nove provjere u metodi za provjeru kolizija `check_collisions()`.

```
def check_collisions(self):
    ...
    # Ako je došlo do kolizije između rakete invadera i
    # jednog od objekta grupe bunkera, briši oba.
    pg.sprite.groupcollide(self.invader_missile_grp,
                           self.bunker_grp,
                           True, True)
```

```
        # Isto kao i gore, samo za raketu igrača.
        pg.sprite.groupcollide(self.player_missile_grp,
                               self.bunker_grp,
                               True, True)
        # Ako dođe do kolizije invadera i bunkera,
        # treba brisati objekte grupe bunkera, a invadera ne.
        pg.sprite.groupcollide(self.invader_grp,
                               self.bunker_grp,
                               False, True)
        ...
```

10.2. Dodavanje zvukova

Sve zvukove igre ćemo učitati u metodi `load_sounds()` klase `Game`. Na sličan način kao i slike, zvukove popisujemo u listu `sound_names`. Koristeći metodu razumijevanja rječnika (eng. *dictionary comprehension*) generiramo rječnik zvukova. Ključ (*key*) rječnika je naziv zvuka u obliku *stringa*, a vrijednost (*value*) je učitani zvuk. Paket Pygame radi s *wav* i *ogg* zvučnim datotekama. Budući da je format *ogg* komprimiran i bitno manje veličine, a ne gubi kvalitetu zvuka (*lossless compression*) odlučili smo sve zvukove pripremiti u tom formatu.

NAZIV ZVUKA
extra_life.ogg[38]
invader_kill.ogg[39]
mystery.ogg[40]
mystery_kill.ogg[41]
player_kill.ogg[42]
player_shoot.ogg[43]
move*.ogg[44]

Tablica 2: Popis svih zvukova igre

Neke ćemo zvukove malo prilagoditi, tj. stišati. Metoda `set_solume()` prima argumente u vrijednosti od 0.0 do 1.0, gdje je 1.0 uobičajena jačina zvuka.

[38] Izvor: https://freesound.org/people/plasterbrain/sounds/397355/
[39] Izvor: http://www.classicgaming.cc/classics/space-invaders/sounds
[40] Izvor: http://www.classicgaming.cc/classics/space-invaders/sounds
[41] Izvor: https://freesound.org/people/Kodack/sounds/258020/
[42] Izvor: https://freesound.org/people/Quaker540/sounds/245372/
[43] Izvor: http://www.classicgaming.cc/classics/space-invaders/sounds
[44] Izvor: https://freesound.org/people/PlanetroniK/sounds/371060/

```
    def load_sounds(self):
        sound_names = ["extra_life", "invader_kill", "mystery",
                       "mystery_kill", "player_kill", "player_shoot"]
        self.sounds = {name: pg.mixer.Sound(f"sounds/{name}.ogg")
                                              for name in sound_names}
        self.main_sound = [pg.mixer.Sound(f"sounds/move{i}.ogg")
                                                  for i in range(4)]
        self.sounds["player_kill"].set_volume(0.25)
        self.sounds["mystery_kill"].set_volume(0.5)
```

Zvukove pokrećemo metodom `play()`, a gasimo metodom `stop()`.

```
    ...
    self.sounds["player_shoot"].play()
    ...
    self.sounds["mystery"].stop()
    ...
```

Glavni zvuk igre, `main_sound`, je lista koja je sastoji od četiri silazno kromatskih tonova istog zvuka: `move0`, `move1`, `move2`, `move3` (indeksi: 0, 1, 2 i 3). Njega kontroliramo u metodi `play_main_sound()`. Svaki put kad se pomaknu *invaderi*, poziva se i metoda glavnog zvuka igre, tj. učitava se i reproducira sljedeći zvuk iz liste. Operatorom *modulo* (ostatak dijeljenja) to postižemo na sljedeći način: na početku je indeks postavljen na 0 iz čega slijedi: *0 % 4 = 0*; odsvira se zvuk iz liste pod indeksom 0. Zatim se indeks poveća za 1. Kad se *invaderi* ponovno pomaknu odsvirat će se zvuk pod indeksom 1 (*1 % 4 = 1*). Zatim zvuk 2 (*2 % 4 = 2*), potom 3 (*3 % 4 = 3*) koji je i posljednji zvuk u listi. Konačno, kad indeks glavnog zvuka postane 4 odsvirat će se opet zvuk iz liste pod indeksom 0 (*4 % 4 = 0*). Na taj način rotiramo uvijek ista četiri zvuka u igri.

```
    def play_main_sound(self):
        self.main_sound_index %= 4
        self.main_sound[self.main_sound_index].play()
        self.main_sound_index += 1
```

Da bi se zvukovi mogli učitavati i reproducirati potrebno je inicijalizirati `mixer` modul. Njega možemo inicijalizirati i samo s `pg.init()` naredbom, ali onda su učitani i njegovi uobičajeni (*default*) argumenti.

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Uobičajeni *buffer* (*međuspremnik*) argument dolazi s vrijednošću 4096 i to uzrokuje preveliku *latenciju*, tj. kašnjenje zvukova. Budući da želimo smanjiti kašnjenje zvukova, moramo promijeniti vrijednost *buffera*, pa ćemo i modul `mixer` inicijalizirati zasebno i prije `pg.init()`.

Buffer smanjujemo na 256, uz mali rizik da se pojavi *droput* zvuka, tj. da se možda ponekad zvuk ne reproducira. Nakon dosta testiranja i iskušavanja, na *bufferu* 256 nismo primijetili niti jedan *dropout*, pa ga stoga ostavljamo na toj vrijednosti.

```
pg.mixer.init(buffer=256)
```

11. NAMJEŠTANJE TEŽINE I STANJA IGRE, INSTALACIJA

11.1. Podešavanje težine igre

Kako bi igra bila zanimljivija i izazovnija postupno će se njeni pojedini elementi otežavati. Ubrzavat će se kretnja *invadera*, nakon svake svladane razine će njihov početni položaj biti niži, te će sve češće ispaljivati rakete.

U konstruktoru klase Game definiramo dva nova atributa. Atributom `level` uvodimo razine, kako bismo sa svakom sljedećom otežali malo igru. Predefinirat ćemo i maksimalnu brzinu kretanja *invadera*.

```
self.level = 1
self.max_invader_speed = 50
```

U metodi `create_invaders()` se svaki put prilikom kreiranja *invadera* provjerava na kojoj je razini igrač. Sa svakom sljedećom razinom *invaderi* će bit postavljeni 20 piksela niže. Dakle, što je razina veća, to je i početni položaj *invadera* niži, samim time i borba s njima je zahtjevnija.

```
def create_invaders(self):
    invaders_pos_y = min(
                INVADER_START_POS[1] + (self.level - 1) * 20, 200)
    ...
```

Na početku igre *invaderi* će biti postavljeni na visini kao dosad.

```
min(INVADER_START_POS[1] + (self.level - 1) * 20, 200) ->
min(100 + (1 - 1) * 20, 200) ->
min(100 + 0 * 20, 200) ->
min(100, 200)
Dakle, visina je 100.
```

Na petoj razini će biti postavljeni na visini 200 i dalje se više neće spuštati s povećanjem razine.

```
Npr.: razina (level) 7:
min(INVADER_START_POS[1] + (self.level - 1) * 20, 200) ->
min(100 + (7 - 1) * 20, 200) ->
min(220, 200)
Dakle, visina je 200.
```

U metodi `create_invader_missile()` ćemo promijeniti uvjet za kreiranje rakete, jer želimo da s povećanjem razine *invaderi* češće pucaju. Tako će, primjerice, na početku igre pucati kao i dosad, na četvrtoj razini dva puta češće, dok će na sedmoj razini pucati čak tri puta češće nego na početku.

```
    def create_invader_missile(self):
        ...
        if num < 7 + self.level * 3:
        ...
```

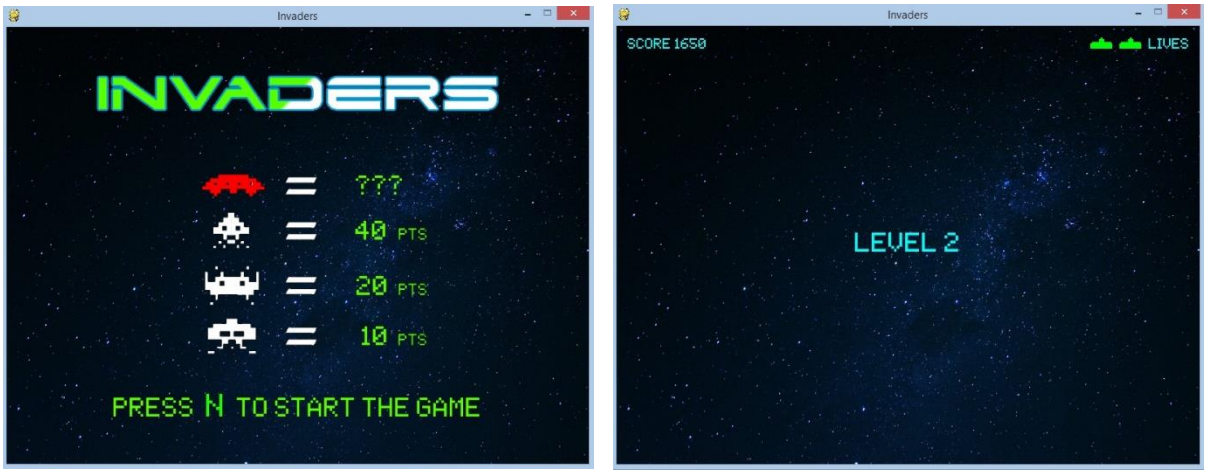
Metoda `process_invaders_speed()` usklađuje novu brzinu kretanja *invadera* parametrom `value`. Što je atribut `invaders_animate_time` manji, to je brzina kretnje *invadera* veća. Svaki put kad pogodimo *invadera* pozvat će se ta metoda i povećati brzina za 10. Na jednak način, nakon svakog silaznog spuštanja *invadera*, kad su na rubu prozora, povećat će se brzina za 25.

```
    def process_invaders_speed(self, value):
        self.invaders_animate_time = max(
                                     self.invaders_animate_time - value,
                                     self.max_invader_speed)
```

11.2. Podešavanje stanja igre

Uz `game_over`, uvest ćemo još dva stanja: `game_started`, stanje koje provjerava je li igra započela ili smo na početnom zaslonu i `game_paused`, stanje igre između dviju razina. Atribut `game_paused_time` će uzeti točno vrijeme kad je igra pauzirana. Zbog jednostavnosti prikaza zaslona između dviju razina kreirat ćemo još jednu grupu, `game_paused_grp`.

Igra će biti pauzirana svaki put kad igrač uspije uništiti sve *invadere*, kao i na početku igre, kad se pritisne tipka '*n*' u početnom zaslonu. Kad je igra pauzirana ispisujemo u sredini prozora razinu koju igrač započinje.



Slika 15: Početni prikaz igre i prikaz igre za vrijeme prekida

Kreiramo metodu `create_game_paused_screen()` za pripremu zaslona za vrijeme prekida igre. Gasimo zvuk *mysterija* i iz `all_sprites` grupe brišemo *spriteove* kako ne bi bilo zaostataka u novoj razini. Stvaramo tekstualni objekt s brojem razine, i dodajemo ga zajedno s ostalim tekstom u novu grupu koja će biti aktivna dok je igra u `game_paused` stanju.

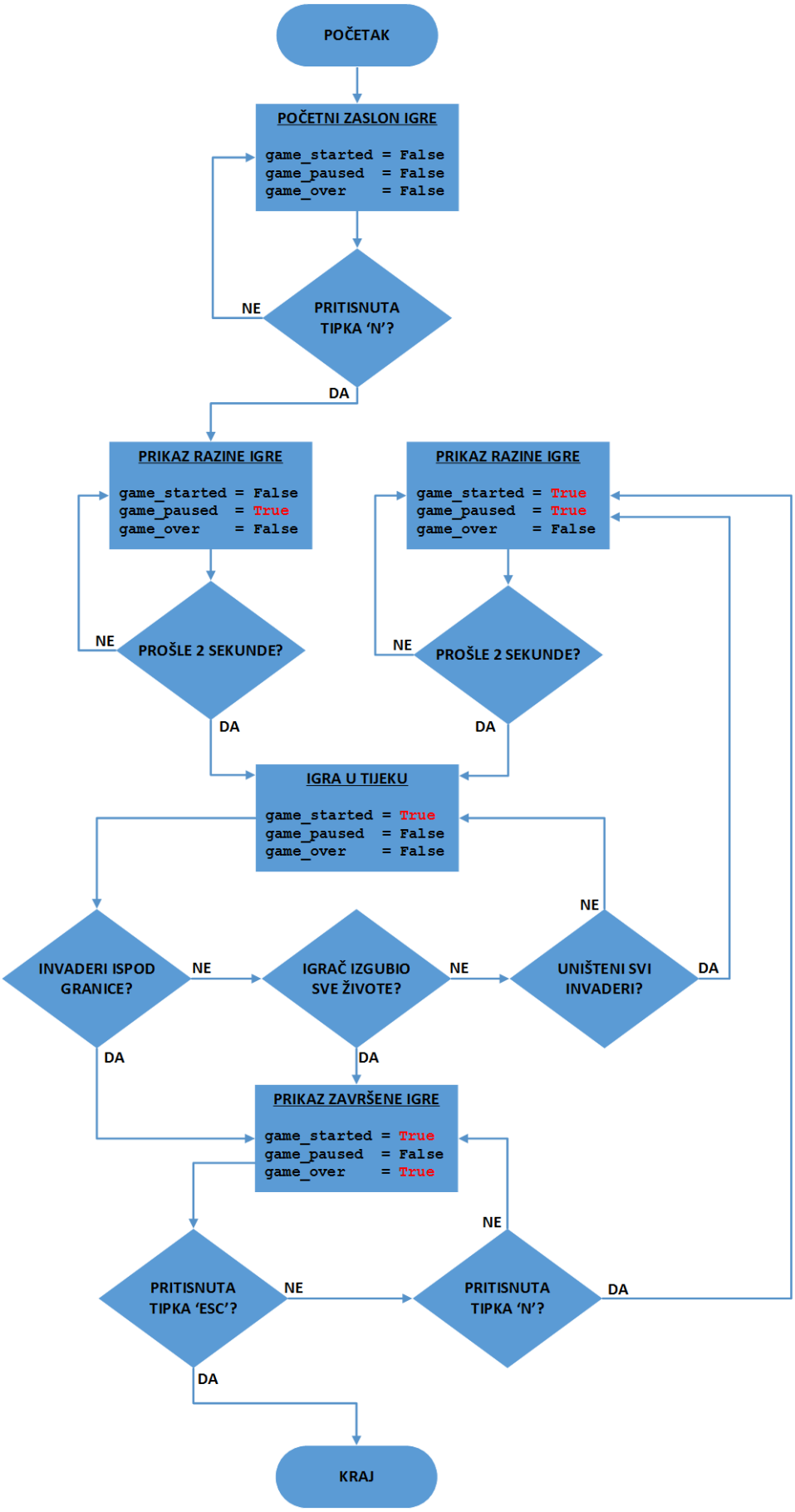
```
def create_game_paused_screen(self):
    self.sounds["mystery"].stop()
    self.all_sprites.remove(self.mystery_grp, self.invader_missile_grp)
    self.mystery_grp.empty()
    self.invader_missile_grp.empty()
    level_str = f"LEVEL {self.level}"
    level_pos = (SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2)
    level_txt = Text(level_str, level_pos, FONT, 30)
    level_txt.rect.center = level_txt.pos
    self.game_paused_grp.add(level_txt, self.text_grp, self.life_grp)
```

U metodi `process_events()` namještamo da igrač može započeti igru pritiskom na tipku '*n*', te onemogućavamo pucanje dok je igra pauzirana.

```
def process_events(self):
    ...
    if event.type == KEYDOWN and event.key == K_SPACE:
        if not self.player_missile_grp and self.player_grp and
                                           not self.game_paused:
            self.create_player_missile()
    if event.type == KEYDOWN and event.key == K_n:
        if self.game_over:
            ...
        if not self.game_started:
            self.game_paused = True
            self.game_paused_time = pg.time.get_ticks()
            self.create_game_paused_screen()
```

Na kraju u metodi za obradu *frameova* `display_frame()` razgranat ćemo izvođenje dijelova igre zavisno o stanju u kojem se igra nalazi.

```
def display_frame(self, screen):
    if self.game_over: ...
    elif self.game_paused: ...
    elif self.game_started: ...
    else: ...
```



Slika 16: Prikaz stanja igre u dijagramu tijeka

11.3. Instalacija igre

Za instalaciju igre Invaders na računalima s Windows operativnim sustavom potrebno je imati instaliran Python (verzija 3.6+). Kako smo već spomenuli instalacijske datoteke su dostupne na poveznici: https://www.python.org/downloads/release/python-370/. Predlažemo instalaciju na način opisan u drugom poglavlju. Nakon instalacije Pythona potrebno je instalirati paket Pygame (verzija 1.9.4) prema uputama iz trećeg poglavlja.

Kad su instalirani Python i Pygame igra se može preuzeti na: https://github.com/jcrljenko/Invaders/releases/download/v1.0/Invaders.zip. Skinuta datoteka se raspakira u željenu mapu i ako se pri instalaciji Python dodao u PATH varijablu dovoljno je pokrenuti *invaders.py* datoteku. Alternativno, igru možemo pokrenuti i iz editora IDLE koji uobičajeno dolazi s instalacijom Pythona. Pokrenemo IDLE, pritisnemo *File/Open* i izaberemo *invaders.py* datoteku iz mape gdje smo ju raspakirali. U novom prozoru koji se otvorio pritisnemo *Run/Run Module* i igra će se pokrenuti.

12. ZAKLJUČAK

Paket Pygame omogućuje relativno jednostavno stvaranje osnovnih elemenata igre, poput prikaza slika, *spriteova*, učitavanja zvukova itd. Jednako tako olakšava nam rješavanje kolizija između objekata te pojednostavljuje kretanje objekata s već gotovim metodama i funkcijama. Nadalje, omogućuje ažuriranje i crtanje više stotina *spriteova* odjednom koristeći grupe. No, želi li se igra pedantnije izraditi potrebno je osmisliti svaki korak, pa u tom smislu Pygame služi kao pomoć, ali ne i kao gotov proizvod. Dobra strana je, kao što smo na početku naveli, da je spomenuti paket dostupan široj zajednici (*open source*), pa ga svi zainteresirani mogu instalirati i koristiti.

U radu smo prikazali postupak nastajanja na prvi pogled vrlo jednostavne igre, ali u kojoj je bilo potrebno osmisliti svaki detalj kako bi bila zanimljivija i privlačnija za igru. Opisali smo svaki korak pri nastanku igre: od početnog postavljanja pozadinske slike, dodavanja letjelice igrača, podešavanja pokreta *invadera* ili *mystery* letjelice, animacija i eksplozija, postavljanja statičkog, dinamičkog i treptajućeg teksta, izrade bunkera pomoću mreže pravokutnih objekata, postavljanja težine igre, zvukova igre te konačno tri stanja igre. Na taj smo način nastojali izradu prikazati pregledno kako bi se mogao slijediti proces postupnog nastanka igre.

13. LITERATURA

1. Barry, P., *Head First Python*, 2. izdanje, Sebastopol, O'Reilly Media, 2017.
2. Budin, L. et al., *Napredno rješavanje problema programiranjem u Pythonu*, Zagreb, Element, 2013.
3. Budin, L. et al., *Rješavanje problema programiranjem u Pythonu*, Zagreb, Element, 2014.
4. Kalafatić, Z. et al., *Python za znatiželjne. Sasvim drukčiji pogled na programiranje*, Zagreb, Element, 2016.
5. Lott, S. F., *Modern Python Cookbook*, Birmingham, Packt Publishing Ltd., 2016.
6. Phillips, D., *Python 3 Object-oriented Programming*, 2. izdanje, Birmingham, Packt Publishing Ltd., 2015.
7. Urban, M., J. Murach, *Murach's Python Programming*, Fresno, Mike Murach & Associates, 2016.
8. Wentworth, P. et al., *How to Think Like a Computer Scientist: Learning with Python 3 Documentation*, 3. izdanje, 2012., dostupno na: http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/, (pristupljeno 4. 9. 2018.)

WEB STRANICE

1. Bennett, J., *An introduction to Python bytecode*, 2018, https://opensource.com/article/18/4/introduction-python-bytecode, (pristupljeno 4. 9. 2018.)
2. *PyPI – The Python Package Index*, https://pypi.org/, (pristupljeno 4. 9. 2018.)
3. Robinson, D., *The Incredible Growth of Python*, 2017, https://stackoverflow.blog/2017/09/06/incredible-growth-python/, (pristupljeno 4. 9. 2018.)
4. *Simple DirectMedia Layer*, https://www.libsdl.org/, (pristupljeno 4. 9. 2018.)
5. *TIOBE Index for September 2018*, 2018, https://www.tiobe.com/tiobe-index/, (pristupljeno 4. 9. 2018.)

PYGAME I PYTHON DOKUMENTACIJA

1. *pygame.display*, https://www.pygame.org/docs/ref/display.html, (pristupljeno 4. 9. 2018.)
2. *pygame.image,* https://www.pygame.org/docs/ref/image.html, (pristupljeno 4. 9. 2018.)
3. *Pygame Intro*, https://www.pygame.org/docs/tut/ImportInit.html, (pristupljeno 4. 9. 2018.)

4. *pygame.Rect*, https://www.pygame.org/docs/ref/rect.html, (pristupljeno 4. 9. 2018.)
5. *pygame.sprite*, https://www.pygame.org/docs/ref/sprite.html, (pristupljeno 4. 9. 2018.)
6. *pygame.Surface*, https://www.pygame.org/docs/ref/surface.html, (pristupljeno 4. 9. 2018.)
7. *pygame.time*, https://www.pygame.org/docs/ref/time.html, (pristupljeno 4. 9. 2018.)
8. *Pygame wiki, FrequentlyAskedQuestions*, https://www.pygame.org/wiki/FrequentlyAskedQuestions, (pristupljeno 4. 9. 2018.)
9. *Python 3.7.0 documentation, os – Miscellaneous operating system interfaces*, https://docs.python.org/3/library/os.html, (pristupljeno 4. 9. 2018.)
10. *Python 3.7.0 documentation, random – Generate pseudo-random numbers*, https://docs.python.org/3/library/random.html, (pristupljeno 4. 9. 2018.)

IZVORI

1. *Freesound – "Arcade Bleep Sound" by Kodack*, https://freesound.org/people/Kodack/sounds/258020/, (pristupljeno 4. 9. 2018.)
2. *Freesound – "C square bass.wav" by PlanetroniK*, https://freesound.org/people/PlanetroniK/sounds/371060/, (pristupljeno 4. 9. 2018.)
3. *Freesound – "Explosion" by Quaker540*, https://freesound.org/people/Quaker540/sounds/245372/, (pristupljeno 4. 9. 2018.)
4. *Freesound – "Tada Fanfare A" by plasterbrain*, https://freesound.org/people/plasterbrain/sounds/397355/, (pristupljeno 4. 9. 2018.)
5. *Outer Space Stars Wallpaper*, https://www.allwallpaper.in/outer-space-stars-wallpaper-16838.html, (pristupljeno 4. 9. 2018.)
6. *Sound Effects and Music from Space Invaders the Classic Arcade Game*, http://www.classicgaming.cc/classics/space-invaders/sounds, (pristupljeno 4. 9. 2018.)
7. *space invaders regular font*, https://fonts2u.com/space-invaders-regular.font (pristupljeno 4. 9. 2018.)

POPIS SLIKA I TABLICA

Slika 1: Glavna programska petlja

Slika 2: Player

Slika 3: Letjelica u zadanim granicama

Slika 4: Dvije slike istog *invadera*

Slika 5: Svi *invaderi* u igri

Slika 6: Animirani *invaderi* konfigurirani po retcima i stupcima

Slika 7: Punjenje grupe *invadera* po retcima

Slika 8: Punjenje grupe *invadera* po stupcima

Slika 9: Spritesheet za eksploziju letjelice

Slika 10: Mystery letjelica

Slika 11: Pogodak *mystery* letjelice

Slika 12: Letjelica igrača sa štitom i bez štita

Slika 13: Završni prikaz igre

Slika 14: Početni izgled bunkera

Slika 15: Početni prikaz igre i prikaz igre za vrijeme prekida

Slika 16: Prikaz stanja igre u dijagramu tijeka

Tablica 1: Popis važnijih Pygame modula

Tablica 2: Popis svih zvukova igre

SAŽETAK

U radu je prikazan postupak nastajanja jednostavne arkadne igre rađene u Pythonu. Za potrebe rada pomoću paketa Pygame izrađena je igra *Invaders*, dostupna na repozitoriju Github: https://github.com/jcrljenko/Invaders.

Cilj rada bio je prikazati postupno građenje svakog elementa igre. Igra je nastajala u nekoliko faza, te je svaka faza u radu detaljno opisana i obrazložena. Na prvi pogled igra je vrlo jednostavna no, kako bi bila zanimljivija i privlačnija za igru bilo je potrebno osmisliti puno detalja. U radu je stoga sve spomenuto obrazloženo, te je opisan svaki korak pri nastanku igre: od početnog postavljanja pozadinske slike, dodavanja letjelice igrača, podešavanja pokreta *invadera* ili *mystery* letjelice, animacija i eksplozija, postavljanja statičkog, dinamičkog i treptajućeg teksta, izrade bunkera pomoću mreže pravokutnih objekata, postavljanja težine igre, zvukova igre te konačno tri stanja igre (*game started, game paused, game over*). Rad je popraćen i oprimjeren mnogim prilozima izvornog koda, kako bi se lakše mogao slijediti proces.

Ključne riječi: Python, Pygame, arkadna igra, Invaders, programiranje

SUMMARY

This paper shows the development process of a simple arcade game made in Python. The game Invaders was also created for this paper, with the help of the Pygame package and is available on the repository Github: https://github.com/jcrljenko/Invaders.

The goal of this paper was to show the systematic building of all elements of a game. The game was developed in several phases; each phase is displayed and explained in detail. It seems fairly simple at first glance but, in order for the game to be more interesting and exciting to play, many details had to be created. Therefore, this paper describes all of the above mentioned, furthermore, all the steps taken towards the creation of a game from adding its initial background image, addition of player's aircrafts, movement adjustments for the Invader or Mystery aircrafts, animation and explosions, the insertion of static, dynamic and blinking text, the development of a bunkers with the help of a web of rectangular objects, setting the skill level of the game, sounds and the three game states (game started, game paused, game over). Several attachments of source code are provided in order to more easily follow the process.

Key words: Python, Pygame, arcade game, invaders, programming