TABLE OF CONTENTS:

GOAL OF THE PROJECT:

The primary goal of this project is to create an engaging and interactive game that combines elements of strategy and skill. The game aims to captivate players by offering a mix of shooting, dodging obstacles, and strategic planning. The interface is designed to be user-friendly, ensuring that it is intuitive and easy to navigate, making the game accessible and enjoyable for a broad audience. Maintaining a smooth and responsive user experience is critical, achieved through efficient rendering, minimal lag, and seamless transitions between different game states and screens.

The game incorporates dynamic and interactive elements such as moving targets, falling meteors, and power-ups to keep gameplay interesting and varied. These elements interact with each other in a realistic manner, enhancing the overall gaming experience. Multifunctional game modes are provided to cater to various player preferences, including a standard mode where players aim to destroy targets and avoid obstacles, and a secondary mode where players survive against increasingly difficult waves of falling meteors.

A system for tracking high scores and displaying leaderboards is implemented to give a sense of competition and achievement among players, encouraging them to share their scores and challenge friends. The game architecture is designed to be flexible and scalable, allowing for easy addition of new features, levels, and game objects, ensuring that the game can be expanded and improved upon in future iterations. Compatibility between different devices is ensured by optimizing the game to run smoothly on various machines and screen resolutions.

By achieving these goals, the project aims to deliver a high-quality game that not only provides an enjoyable and challenging experience for players but also serves as a showcase of effective game development practices using the Kivy framework and Python programming language.

ASSUMPTIONS:

-No major assumptions regarding the user are postulated, the game is designed in order to be played by everyone, even without previous experiences in gaming, and to be played on every computer device, regarding the resolution.

-It is assumed that future developers, in case they want to edit the code, have an understanding of Python and the Kivy framework.

METHODS:

-implementation of simplified physics for objects movement

-use of the Kivy framework for game mechanics and user interface

-use of object oriented programming to grant modularity

PROCEDURES:

-import of the necessary libraries

-creation of the various game objects such as player, projectiles, obstacle and their relative dynamics

-implementation of the game objects in the two different modalities, tailoring the mechanics for the specific use

-creation of the user interface, such as main menu, leaderboard screen, login screen, modality selection, help screen, option menu.

-creation of a smooth switch method in order to navigate easily between screen

SUCCESS CRITERIA

-implement the screen and frame rate

-implementation of the following game objects and their specifics: Bullet, Bombshell, Laser, Physics, Rock, Perpetio, Bulletproof Mirror, Hall of Fame, Save/Load, Help Page, Target.

Note on Success Criteria:

Project success criteria have been slightly changed by the author of this project in order to fit better in the overall production. The Perpetio object, although present codewise, has not been implemented in the game. The Bombshell has been substituted by the Meteor, the Extinction game modality has been introduced, the Hall of Fame works on the score of the Extinction Modality, Save/Load functionality has been implemented through a Login/Logout system, a Boss battle has been introduced, creation of an Option Menu and Modality Selection menu, Player Movement, different types of Targets. It is believed that those changes would not drastically alter the nature of the Cannon Game as intended, since all the main functionality are preserved.

MOTIVATION BEHIND DESIGN CHOICES:

The graphical design choices are mainly motivated by the passion for dinosaurs of the author.
The design is intended to be captivating and engaging for the user base, granting a fun experience while maintaining a clear and easy to navigate interface.
The game is divided in two different modalities: In the Main Game the objective of the user is to defeat the Dinocyborgs through different levels, while in the secondary modality, called Extinction, the objective of the user is to survive as long as possible under a rain of meteorites.
User experience is prioritized through an easy to navigate Menu and a Login-Logout mechanism that allow the user to receive a personalized experience of the game, since is it possible for the user to select an unique username and avatar picture.
The scores obtained by the user in the Extinction modality will be displayed in the Leaderboard Screen, easily accessible from the Main Menu or after completing a game in the Extinction modality.
To make the experience more appealing for the user, custom music themes have been introduced into the game, although the option to deactivate them is given in the Option Menu through a volume slider.
A Help menu with all the necessary information is accessible from the Main Menu to solve all the user doubts.

LOGICAL STRUCTURE OF THE SOLUTION:

1) INITIALIZATION:

The initialization of the game is managed by the MyApp (App) class, which starts by checking if a user is already logged-in and then proceeds to create the necessary screens, such as the Login Screen of Main Menu Screen using Kivy ScreenManager.

The MyApp class is also used to regulate the switch between different screens throughout the use of a logic where not useful screens get eliminated. The class also administers general music volume.

Each Screen is hence created when needed and eliminated when not useful anymore.

SCREENS:

Is it possible to navigate screens throughout the use of predisposed buttons, all screens are accessible from the Main Menu. the screens are the following:

LoginScreen: manage login mechanics

MainMenu: serves as an entry point for every other screen

OptionMenu: gives access to the leaderboard screen, logout system and volume management

ModalityMenu: allow to select the game modality

HelpScreen: contains all necessary information to play the game

GameScreen: host the primary modality

Estinzione: host the Extinction modality

Top3Screen: contain the leaderboard

GAME OBJECTS:

The game objects are structured through the use of classes in order to grant modularity and organization, except for the rock-perpetio case, every game object has its own class.
If no repetition happens in the code, methods (as bullet movement) are directly implemented in the code and not in the class.

2) GAME MECHANICS:

Game mechanics depend on modality , although some are in common.

PHYSICS SIMULATION: The physics of the game try to reproduce the real world one, although simplified. The trajectory of the projectiles (Bullet and MeteorCaller) follow projectile motion, while the Laser follows uniform rectilinear motion.

OBSTACLES: The obstacles differ by modality, in the Main modality the obstacles are the Rock (stops the laser and bullet, can be destroyed by the meteor) and the Mirror (stop the bullet and reflect the Laser following Snell's law), while in the Extinction modality the only obstacle consist in deadly meteors falling from the sky.

PROJECTILES: The available projectiles differ by modality. In the Main modality the available projectiles are the Bullet (parabolic trajectory) , the Laser (multicolor, linear trajectory) and the Meteor Caller (replace the Bombshell, parabolic trajectory, less range than bullet, attract a meteor on the landing spot), while in the Extinction modality the only available projectile is the Laser.

TARGETS: The targets differ by modality. in the Main modality the target are the Pterodactyl (flying moving target), the Velociraptor (stationary target) and the Boss Target (available only during boss battle, consist in weak points that appears on the boss. when hitted, boss health drops.)

PLAYER: The player interacts with the game throughout a pink tank, is it possible for the player to move and to rotate the muzzle of the tank.

3) USER INTERFACE:

The user interface has been structured in order to be intuitive and easy to use. The game is designed to be played throughout the use of a keyboard and a mouse/touchpad. The hotkeys differ by modality and are explained in the HelpScreen.

4) OTHER FUNCTIONALITIES:

LOGIN/LOGOUT: Everytime the game is started, the LoginManager checks if a user is logged-in. If it is, the game directly opens on the Main Manu and the logged-in user data gets uploaded, if it is not, the Login Screen will be the first to be visualized. If the inserted username is not already present in the user database (leaderboard.txt) the user will need to select an avatar to complete his profile, if it is already present, the avatar selection phase will be skipped. Every time a user log-in/log-out , the login session (login_session.txt) gets updated with the state of the log-in (true if a user is logged-in, false otherwise) and with the username on the logged-in user (blank if no user is logged-in).

LEADERBOARD: The Leaderboard functions keep track of the user scores in the Extinction modality. The user scores are kept track of in the user database (leaderboard.txt) which consist of a triple composed by the following data points,username,avatar-image. The list of users is sorted by points, with the top-score user at the top. In the Top3Screen, the 3 top-score user data gets to be displayed.

SOUNDS: Sounds are handled through the use of Kivy SoundLoader.

5)MODALITIES:

The game is divided in two different modality, the Main Game and the Extinction mode, each modality has his own characteristics and game loops.

MAIN GAME: It is closer to the goal of the project and consists of an artillery game. Contains the big part of the game objects. The game loop handles checking for collisions ,user input detection, object motion.

EXTINCTION: In this modality the only objective is to survive, giant meteors fall from the sky. The game loop handles user input, laser movement, while the Apocalypse function handles the falling of the meteors and collisions.

IMPLEMENTATION:

In this section only few examples of the implemented code will be discussed, to get a full grasp of how the code works check the comments in the code file.

COLLISIONS:

```python
def collides(rect1, rect2):
    # Extract the top-left corner and dimensions of both rectangles
    r1x, r1y, r1w, r1h = rect1[0][0], rect1[0][1], rect1[1][0], rect1[1][1]
    r2x, r2y, r2w, r2h = rect2[0][0], rect2[0][1], rect2[1][0], rect2[1][1]

    # Check for no overlap
    if (r1x + r1w <= r2x) or (r2x + r2w <= r1x) or (r1y + r1h <= r2y) or (r2y + r2h <= r1y):
        return False

    else:
        return True
```

This custom collision function extracts the coordinates of the Rectangles (game objects) and checks if there is an overlap.

```python
if collides(((self.laser.laser_translation.x, self.laser.laser_translation.y), (0, 50)), (self.boss.target.pos, self.boss.target.size)):
```

The collides function takes as arguments the position and the size of the two objects of which the collision is checked.

LOGIN SESSION:

```python
def read_login_session():
    if not os.path.exists(login_session):
        return {"logged_in": False, "username": ""}
    with open(login_session, 'r') as file:
        data = file.read().strip().split(',')
        return {"logged_in": data[0] == 'true', "username": data[1]} if len(data) > 1 else {"logged_in": False, "username": ""} #Check if a user is logged in

def write_login_session(logged_in, username=""):
    with open(login_session, 'w') as file:
        file.write(f"{str(logged_in).lower()},{username}")
```

Those two functions serve the purpose to read and edit the login session file, while the LoginManager class plays an important role in utilizing the extracted data.

```python
class LoginManager:
    def __init__(self):
        self.session = read_login_session()

    def is_logged_in(self):
        return self.session["logged_in"]

    def get_username(self):
        return self.session["username"]

    def login(self, username):
        self.session["logged_in"] = True
        self.session["username"] = username
        write_login_session(True, username)

    def logout(self):
        self.session["logged_in"] = False  #Set the values of the Login state to default (no-one logged in)
        self.session["username"] = ""
        write_login_session(False)

    def user_exists(self, username):
        leaderboard = read_leaderboard(get_file_path('leaderboard.txt'))
        return username in leaderboard

    def add_user_to_leaderboard(self, username, image_path):  #add user to the leaderboard if not already present
        leaderboard = read_leaderboard(get_file_path('leaderboard.txt'))
        leaderboard[username] = {'points': 0, 'image_path': image_path}
        write_leaderboard(get_file_path('leaderboard.txt'), leaderboard)
```

LEADERBOARD:

The Leaderboard functions , or the user data management system, read and edit the file leaderboard.txt, where users data are stored. The mechanism for reading and writing are similar to the login-sessions ones.

```python
def update_leaderboard(file_path, name, points, image_path):
    leaderboard = read_leaderboard(file_path)

    # Check if the name already exists and compare points
    if name in leaderboard:
        if points > leaderboard[name]['points']:
            leaderboard[name]['points'] = points
            leaderboard[name]['image_path'] = image_path
    else:
        leaderboard[name] = {'points': points, 'image_path': image_path}  # Add new entry if the name does not exist

    write_leaderboard(file_path, leaderboard)
```

The update_leaderboard function edits the points of an already existing user if the new score is bigger than the old one.

CLASS EXAMPLE:

```
class Pterodactyl(Widget):
    def __init__(self, **dinobros):
        super().__init__(**dinobros)

        self.GoRight = True
        self.GoDown = True

        with self.canvas:
            self.ptero = Color(1, 1, 1, 1)
            self.ptero = Rectangle(size=(300*ww, 300*ww), pos=(100*ww, 1200*wh), source="./img/ptero.png")

    def fly(self): #Make the enemy move from left to right with sloght up and down movement
        if self.ptero.pos[0] < Window.width - 300*ww and self.GoRight:
            self.ptero.size = (300*ww, 300*ww)
            self.ptero.pos = (self.ptero.pos[0] + 5, self.ptero.pos[1])
        else:
            self.GoRight = False
            self.ptero.size = (-300*ww, 300*ww)
            self.ptero.pos = (self.ptero.pos[0] - 5, self.ptero.pos[1])
            if self.ptero.pos[0] < 300*ww:
                self.GoRight = True

        if self.ptero.pos[1] > 1100*wh and self.GoDown:
            self.ptero.pos = (self.ptero.pos[0], self.ptero.pos[1] - 1)
        else:
            self.GoDown = False
            self.ptero.pos = (self.ptero.pos[0], self.ptero.pos[1] + 1)
            if self.ptero.pos[1] > 1250*wh:
                self.GoDown = True
```

This is an example of a Widget class, it contains the characteristics of the object (as dimension, position, color, image source) and its unique functions. All other Widgets are structured in a similar way.

PROJECTILE ACTIVATION:

```
def activate_caller(self): #Activate the caller, set position
    if not self.caller_active:
        self.caller_active = True
        self.caller_mass = 2.5
        self.caller_velocity = math.sqrt(0.8*ww* self.powerbar.powerbar.size[0] / self.caller_mass)
        self.caller_start_time = Clock.get_boottime()
        cannon_tip_x = self.player.pos[0] + self.player.size[0] / 2
        cannon_tip_y = self.player.pos[1] + self.player.size[1] / 2 + self.cannon.size[0] / 2
        self.mc.set_pos(-self.cannon.size[0] / 2 + cannon_tip_x + self.cannon.size[1] * math.cos(math.radians((self

        #Calculate velocity base on cannon rotation
        self.caller_velocity_x = self.caller_velocity * math.cos(math.radians((self.cannon_rotation.angle) + 90))
        self.caller_velocity_y = self.caller_velocity * math.sin(math.radians((self.cannon_rotation.angle) + 90))
```

```
if keycode[1] == "m" and not self.caller_active and not self.meteor_active:
    self.activate_caller()
    self.mc.caller_sound.play()
```

The projectile's activations are regulated by their “activate” functions. When the input is received the function is called.

THREADING:

```
def apocalypse_thread(self):  #set the threading to not make the game explode
    while self.Ext_mode and not self._stop_event.is_set():
        if len(self.meteors) < MAX_METEORS:
            self.schedule_apocalypse()
        time.sleep(self.meteor_interval)
        #print("thread")
```

Threading is used to schedule the Apocalypse function to make to avoid conflicts.

SCREEN MANAGEMENT:

```
def switch_screen(self, screen_name): #Switch Screen Function, create the screen
    if screen_name not in self.sm.screen_names:
        if screen_name == 'game':
            self.sm.add_widget(self.game)
        elif screen_name == 'menu':
            self.sm.add_widget(self.main_menu)
        elif screen_name == "login":
            self.sm.add_widget(LoginScreen(self.login_manager, name='login'))
        elif screen_name == 'est':
            self.extintion_menu = Estinzione(name='est')
            self.sm.add_widget(self.extintion_menu)
            self.extintion_menu.background_music.volume = self.volume
        elif screen_name == 'option':
            self.sm.add_widget(OptionMenu(self.login_manager, name='option'))
        elif screen_name == 'modality':
            self.sm.add_widget(ModalityMenu(name='modality'))
        elif screen_name == 'leaderboard':
            self.sm.add_widget(self.leaderboard_menu)
        elif screen_name == 'help':
            self.sm.add_widget(self.help_screen)

    if screen_name in ['game', 'est', 'leaderboard']: #stop the main theme when in one of those screen
        self.main_menu.background_music.stop()
    else:
        if not self.main_menu.background_music.state == 'play':
            self.main_menu.background_music.play()

    self.sm.current = screen_name

    # Remove all other screens except the current one
    for screen in self.sm.screens[:]:
        if screen.name != screen_name:
            self.sm.remove_widget(screen)
```

The screen management is regulated in the MyApp Class, the switch_screen function ensures that only one screen at a time is active, in this way conflicts between input handling are avoided and every screen gets rebooted automatically, since it gets created only before access. The class also make sure that the Main Theme “Your Jurassic Era” Plays in the intended Screens.

TESTING AND RESULTS:

Testing was conducted very carefully throughout the whole development of the project.
The testing was conducted on a Huawei Matebook 16s.
User interface got tested by different players, helping in this way to create a clear interface accessible to everyone.
Game design choices were tested by different players, helping regulate difficulty and helping create an entertaining game.
Performancewise, the Main Game found no critical problems, while the Second Modality (Extinction) did not fully pass the testing phase, since performance dropped significantly in an advanced stage of the game.

POSSIBLE IMPROVEMENTS:

In order to improve the game, there are few things that could be added and adjusted.
LEVEL CAMPAIGN:A level campaign could be added to the Main Game, more types of obstacles and projectiles could be unlocked through the levels.
PERSONALIZATION: Even more user personalization could be added to the game, such as the selection of tank colors
POWER UPS: Power Ups in the Extinction mode are nearly ready though not implemented, their implementation would make the modality even more engaging.
STABILIZATION: The Extinction modality did not fully pass the stability testing at advanced stages, it is important to stabilize it before proceeding further.
BOSS BATTLE: The Boss Battle is not yet complete, some abilities as Telekinesis (redirect user projectile) can be added to the boss to make the battle more engaging.

USED TOOLS AND RESOURCES:

Kivy Framework (2.3.0): used for GUI and game mechanics.
Python (3.11.9) : programming language used for implementation

PYTHON LIBRARIES: Math (for use of math functions as sin and cos); Random (to generate random numbers); Threading (avoid conflicts in the Apocalypse function through concurrency); Time (for functions using time); Os (to interact with the os).

REFERENCES:

IMAGES:
TankV1.png : drawn by Sara Ruggeri
Goma.png : drawn by Sara Ruggeri
Cannons.png : drawn by Sara Ruggeri
Dinosfondo.jpg: drawn by Annalisa Rigamonti

All the other images have been generated through OpenAI Dall-E 3 with a custom prompt.

MUSIC THEMES:
All the musics have been generated through suno.com with custom prompt
Jurassic Groove.mp3: Lyrics written by Samuele Pittore

SOUNDS EFFECTS:

The sounds have been collected from various free copyright source and edited with Audacity
impatto.mp3 : https://www.youtube.com/watch?v=CJpwPbA7W-g
meteor.mp3: https://www.youtube.com/watch?v=CJpwPbA7W-g
caller.mp3: A call to Arms - Blizzard (use available for non-commercial educational purpose).
roar.mp3: https://www.youtube.com/watch?v=Ml5ABdE1HtM

LEARNING MATERIAL:

"Kivy: Cross-platform Python framework for GUI apps development." [Link](https://kivy.org/)
Kivy GitHub." [Link](https://github.com/kivy/kivy)
Abdullah, “Python Kivy Game Tutorial.” [YouTube Playlist](https://www.youtube.com/playlist?list=PLMgDVIa0Pg8VP1XqOexsdYP1FralBJP1l)
Amanda Hogan, "Kivy Basics." [YouTube Video](https://www.youtube.com/watch?v=3GBNMBhm6UU)