Motion Flow

CMPT 365 Final Project By
Kier Lindsay (301276300)
Rafid Ashab Pranta (301320738)

Introduction

Motion flow is a game in which, we attempt to create an interesting arcade game from scratch and experimented to control the game using Computer Vision. In the game, The player skis down a generated slope and the objective of the game is to travel farthest and score as high as possible before the sun sets.

Motivation

We decided to create an arcade game because we wanted to experiment making a small game and to find some of the obstacles and difficulties associated with making games especially the physics behind the game. Arcade games are one of the major genres of games where people sit idly and play the games with little to no movement. We wanted to create an arcade game that can be more physically engaging.

Furthermore, using computer vision that detects human motion we can replace the cost for expensive sensors and controllers that are used currently in controlling games using motion sensors. For example, Nintendo Wii , Xbox Kinect and PlayStation Move. Whereas, to utilize computer vision all we need is a webcam and possibly a graphics card for better game play.

Most importantly, our goal was to explore and demonstrate the possible use of Computer Vision and Machine learning in arcade games.

Related Works

OpenPose represents the first real-time multi-person system to jointly detect human body, hand, facial, and foot keypoints (in total 135 keypoints) on single images. We used the pre-trained network provided by Openpose in our project to detect human pose and human motion. (https://github.com/CMU-Perceptual-Computing-Lab/openpose)

Technologies Used

The game is build with in C++ with SFML Library, Thor Library and ImGui Library on a Linux Machine. Cmake was used to generate appropriate platform-specific build scripts which automate building the game in CLion IDE.

The pose detection of the player is achieved with using OpenPose model created by CMU-Perpetual-Computing-Lab, OpenCV libraries. The required computation power was satisfied by Google Cloud Services Virtual Instances.

We used github to communicate and code effectively in a team of two programmers. Our github repository : https://github.com/syonfox/Motion-Flow. We have edited Creative Common audio files to suit the needs of our games using Audacity. We have gimp to initially design the UI and create/edit sprites for the game.

Project Descriptions

The game allows a player to ski down a slope and upon making a smooth landing on the slope of the terrain the player gains speed. The goal of the game is to get the highest score before the sun sets.

The player can get points in three ways :
1. Points for distance traveled
2. Bonus point for each perfect landing
3. Bonus point for consecutive perfect landings (Combos)

There are two different types of input for the game :

1. Keyboard Controls : Pressing S on the keyboard speeds of the player when the player is on the ground and pulls the player downwards when the player is on the air. Pressing P will pause the game and resumes the game.

2. Webcam / Camera : The camera takes a live video of the player and detects key points of the player's body. If the player's shoulders are below a certain horizontal line, it is equivalent of pressing S on the keyboard. If the players shoulder is above a certain it is equivalent of releasing the button S on keyboard.

Terminal Condition : The game ends when the sun sets / the sky turns black which is indicated by a sidebar and the sky.

Implementation of the Game features

For The game we built up the engine in SFML/C++. There are a few major components that were needed to create the game engine. The First major thing that was needed is the main game loop.

Timing: (`main.cpp`)

The loop Runs based on a tick system so we can set the tick rate and frame rate independently. This is so that we can lock the time simulated in the game every tick, this is important so that the game calculations are reproducible and consistent across computers. It also allows us to change the frame rates so that on slower computers we could skip costly render steps while maintaining consistent physics in the background. This timing loop Is used to drive the Engine, it calls then engines `update(deltaTime)` and `render(window)` functions as well as passing any events to the Engine.

Engine Design:(`engine.cpp engine.hpp`)

The Engine is what holds the game together It holds all of the game objects and is responsible for correctly updating the game mechanics and drawing. The engine also has a game state system which allows switching between various hame states such as Paused, Playing, Main Menu, Game Over. When update and render functions are called the engine checks the game state and performs the appropriate actions.
In order to render and update everything we needed to split it up the drawing calls to the various game objects. We also need some signals in order to do things such as stop game sound when paused and reset the game state. Then engine and each game object have a consistent structure in order to keep things more organized.
`update(deltaTime)` - This is called every tick and is meant to do game state and physics updates.
`render(window)` -This is called every frame and is responsible for both drawing stuff with SFML as well as ImGui. These are often separated out into private draw(window) and gui() functions.
`pause()` - This is called any time the game stops and can be used to pause clocks sounds or other things that my run independently of the update function.
`play()` - This is called when the game starts again and can be used in the same way as pause.
`restart()` - This is called when the game wold state needs to be reset and is used to clean up any resources and update variable to the initial conditions.

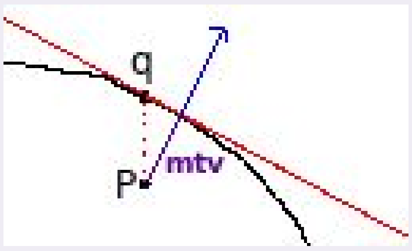
Each function is also responsible for signaling any children they have when these functions are called.

Slope Generation / Collision detection: (`slope.cpp slope.hpp`)

We found it convenient to use a predefined function as the slope. This allows us to do very quick collision detection and Makes it easy to change the level and it would be easy to add new levels to the game or even create the function based off of a random seed. In order to draw the slope and add sprites to it we used a double ended queue. We then countiuly add polygons/sprites for to the ground in front of their queue until the frontmost object is in front of the player by a set buffer distance. Similarly we delete any object that is too far behind the player from the queue.

Because we use have a function as our ground we can easily check if a collision occurs by simply checking if the point is below the curve. If it is then we can approximate the Minimum Translational Vector (mtv) needed to solve the collision by finding the intersection between the line tangent to the curve and the line perpendicular to the tangent which passes through the point we are checking. This may produce a slight over approximation if the point is penetrated deep into the curve but for our use case one iteration of this is enough. The mtv is what is used to calculate the normal force that needs to be applied to the player.

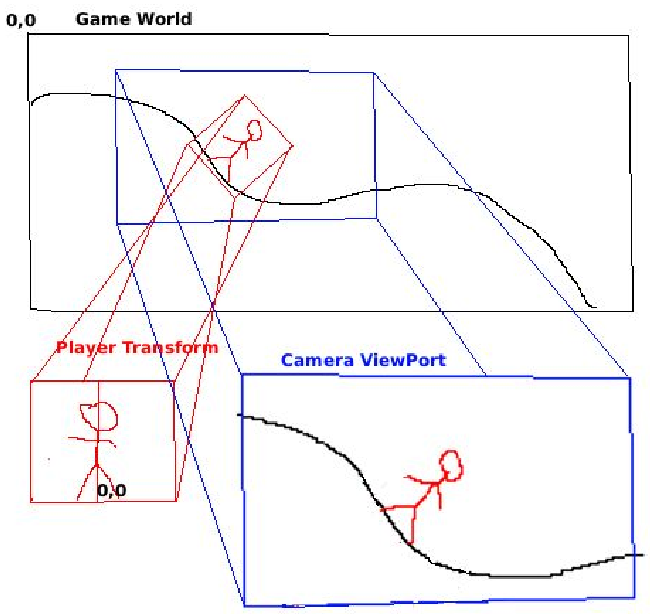
```
bool Slope::colisionPoint(sf::Vector2f p, sf::Vector2f &mtv){
   sf::Vector2f q(p.x, slopeFunction(p.x));
   if(p.y<=q.y) return false; //no collision
   double m = slope(p.x);
   if(fabs(m) < 0.01){ //
       mtv.x = 0;
       mtv.y = q.y - p.y;
   } else {
       sf::Vector2f p2 = p + sf::Vector2f(1, 1/m);
       sf::Vector2f q2 = q + sf::Vector2f(1, -m);
       mtv = lineLineIntersection(p,p2,q,q2) - p;
   }
   return true;
}
```



Game Physics: (`player.cpp player.hpp`)

Since the player is the only physics object in the world it handles its own physics calculations. We base the physics of reality so the player is influenced by applying forces to the player object each tick. These can be external if we wanted another object to push on the player as well as internal for things like gravity. Each tick all control forces are applied as well as gravity. Then it checks if there is a collision and calculates the normal force needed before calculating air drag and finally updating the current velocity and position using the standard kinematic equations.

Game Drawing:

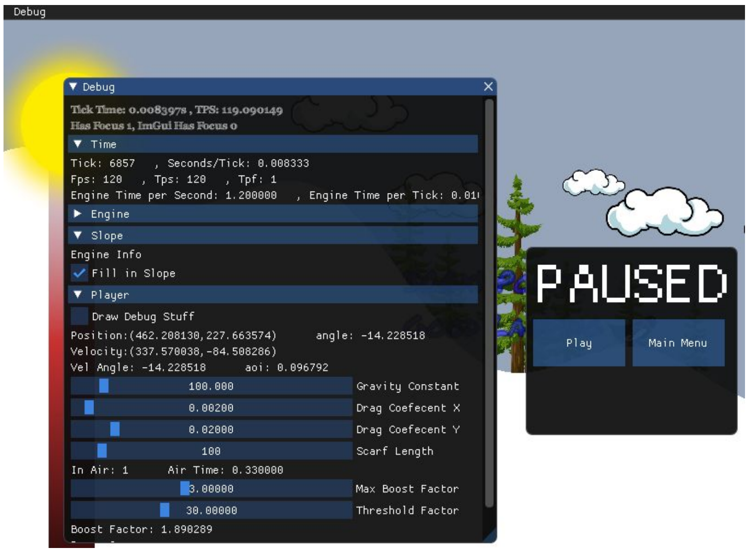


I would like to discuss some of the interesting things we used for drawing We draw everything onto a “Game World” either directly in the case of the slope as the slope function is in game world coordinates.

We also use a transform from player coordinates to the world coordinates. This player transform is what is updated rather then all of the sprites. We can also use this transform to get the world position needed for the landing text and other things relative to the player. In order to adjust the camera we use a viewport into the game world which can be scaled and moved around with the player.

GUI:

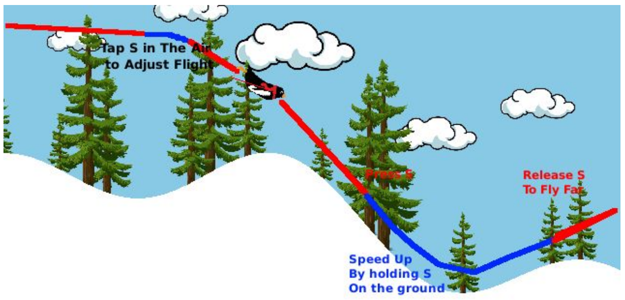
For our debug Gui as well as to implement some quick menus we used ImGui this let us quickly implement nice UI windows and settings. With a little bit of style tweeks we could improve The look of the menus or try to make our own in SFML. Over all Imgui is an excellent tool and allowed us to spend more time working on the game itself. Because it of its design you can insert gui code anywhere in your program easily. The debug gui was essential for playing with the game constants so that we could get a fun game in the end.



```
Debug
▼ Debug
Tick Time: 0.0083978 , TPS: 119.090149
Has Focus 1, ImGui Has Focus 0
▼ Time
Tick: 6857    , Seconds/Tick: 0.008333
Fps: 120    , Tps: 120    , Tpf: 1
Engine Time per Second: 1.200000    , Engine Time per Tick: 0.01
▶ Engine
▼ Slope
Engine Info
✓ Fill in Slope
▼ Player
  Draw Debug Stuff
Position:(462.208130,227.663574)      angle: -14.228518
Velocity:(337.570038,-84.508286)
Vel Angle: -14.228518      aoi: 0.096792
            100.000            Gravity Constant
            0.00200            Drag Coefecent X
            0.02000            Drag Coefecent Y
            100                Scarf Length
In Air: 1      Air Time: 0.330000
            3.00000            Max Boost Factor
            30.00000           Threshold Factor
Boost Factor: 1.890289
```

Game Mechanics:

The core Game mechanic is the control. It is only one Button “S” and if you press it in the air you will slow down and be pushed down towards the ground. Once on the ground You get a Acceleration force applied if you are holding S. You then Want to release S after You leave the ground as you don't glide very well while holding S. The second Core mechanic is Landings the quality of a landing is determined by the players velocity angle relative to the ground. If you hit the ground at a steep angle you receive a penalty to you velocity but if you land with your velocity vector close to parallel with the slope you receive a speed boost on landing. This is very rewarding when you get a perfect landing.



The game ending mechanic is the sun. As it goes down the sky gets darker and darker turning from blue to red to black. When it is night You lose, but the interesting thing is that you can run away from the sun by increasing your speed. We use a threshold speed and if you go faster then it the sun will go back up proportionally to the difference between your speed and the threshold.

```
        sunChange = deltaTime * (playerSpeed.x-threshold)/threshold
```

This creates a race between you and the game. IN order to increase the difficulty as time passes this threshold is slowly increased until eventually you cant maintain a speed fast enough to outrun the night.

Implementation details of Pose Detection

We used the MPII model provided by OpenPose by CMU-Perceptual-Computing-Lab. The model detects 15 key points of the human body.

Which are the following :

```
Head - 0, Neck - 1, Right Shoulder - 2, Right Elbow - 3, Right Wrist -
4, Left Shoulder - 5,
Left Elbow - 6, Left Wrist - 7, Right Hip - 8,Right Knee - 9, Right
Ankle - 10, Left Hip - 11,
Left Knee - 12, Left Ankle - 13, Chest - 14
```

The point of interest in our program are Point 1(Neck), 2(Right Shoulder) and 5(Left Shoulder). This encompasses the head neck and both shoulders. The pose detection program checks if the Point (1,2,5) are below a certain horizontal line.

It is possible by loading the caffemodel provided by openpose using openCV in a net variable, which stores the convolutional neural network provided by the model.

Initially, we took a video recording to detect our poses in every frame of the video recording. The video was taken in 30 Frames Per Second (FPS) in 680 x 480 pixel resolution. Every frame of the video was fed into the OpenPose network in the following process:

The input frames were scaled down to 360 x 248 Resolution. The input frames are then changed in a blob using OpenCV blobFromImage() function and the blob is passed to the network to be analyzed. The frame size dimensions have to be a multiple of 8, which is a requirement of the MPII model.

The convolutional neural network then performs a forward pass on the network with the input image then image is also sized down to a 46 x 46 image within the network.
and outputs 46 x 46 confidence maps for all points defined by the MPII model. If all 15 points are present in the image there will be 15 confidence maps of size 46 x 46

The confidence map of a point is then searched for coordinate with the highest confidence value for a particular point and the coordinate/index of the highest value is stored an array of <cv::points> if they exist. Then we map selected coordinates of the 46 x 46 image to the input size of the image and draw then on that image. Since the points are defined with

respect to a 46 x 46 image they need to be mapped back to the image of 360 x 244 before being drawn. We repeat the process for every point defined by the MPII model.
The related points are connected as seen in the image below to test the output. The points array is then used to detect if the shoulder of the player is below or above the pre defined horizontal line. This process is run every frame. on a seperate thread.

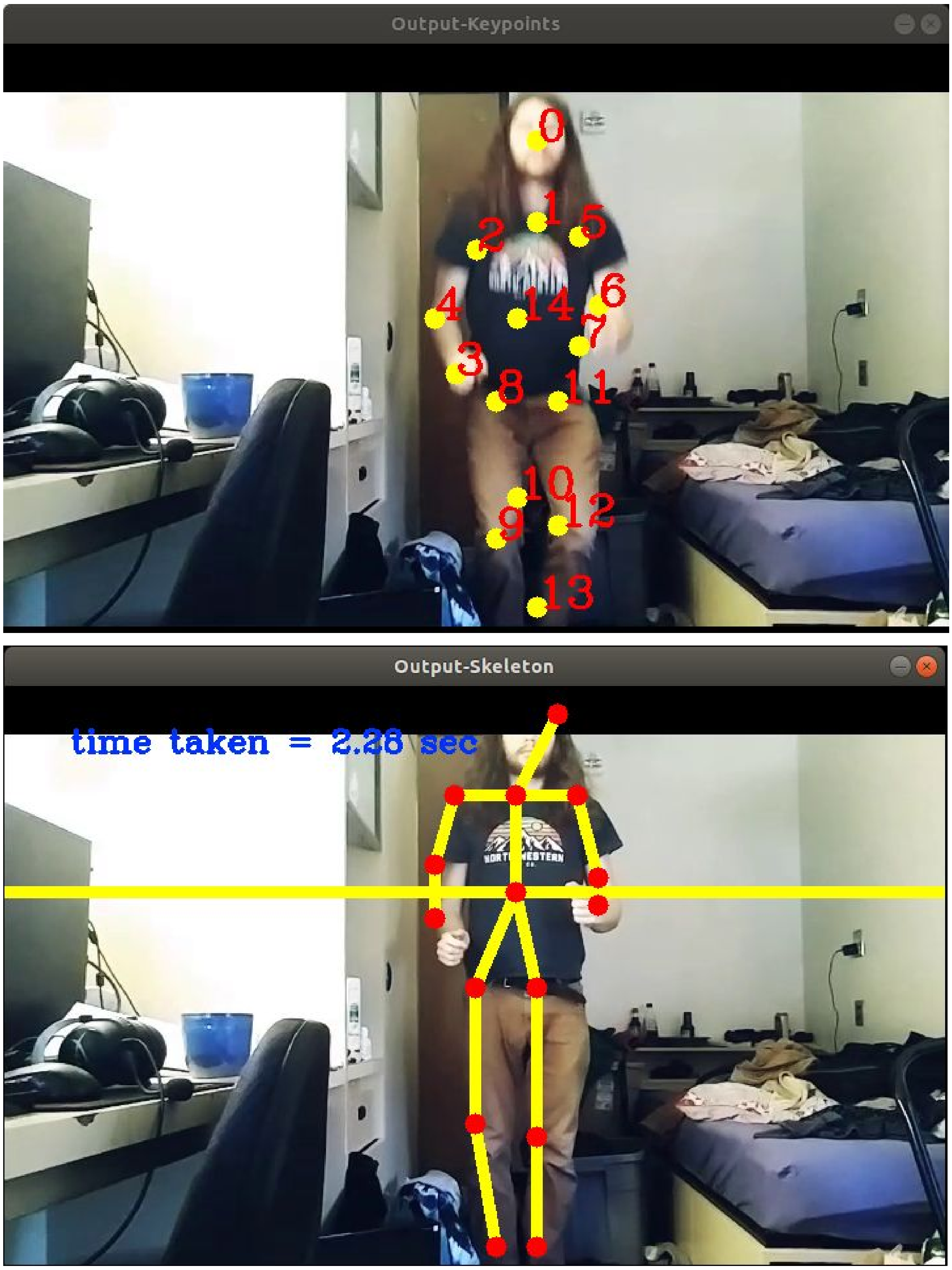


Figure : Output of the Neural network for everypoint drawn on the frame

Multimedia Concepts Used:

1. We used UI design principles to make a fun interactive game and had multiple user test the game.
2. We tried to match the sound design with the feel and look of the game and to enhance the game mechanics.
3. We used Image capture and processing to get inputs for the machine learning model
4. We used Image transformations and manipulations when rendering the game
5. We experimented with different interactive methods for controlling the game. (Pose Controlled and KeyBoard)

Using Pose Detection in the Game :

Initially we have divided the work between ourselves in 2 parts.
1. The game
2. The motion detection

Kier Lindsay worked on the blueprint design of the game and implementation of the engine and player object that dictates the game.

Rafid Ashab Pranta was working on researching motion detection models and detecting two particular poses from a video, which are standing and crouching. The code was written on motion.cpp and motion.hpp files.

In order to merge pose detection with the game we start the pose detection on a seperate thread. This thread constantly runs the image from the webcam through the Network and checks for the players pose. It then stores this and offers a public accessor method for other classes to use. By running in a separate thread we stop the slower Motion Detection process from blocking the main game loop. The game can then access the current player pose at any time by reading the current pose using getPose(). Kier’s player.cpp calls the getPose function from motion every tick and updates the game accordingly.

Even though, initially we started working on two different parts of the games we were very easily able to combine our code. Later, we have further improved the game graphics and efficiency of the pose detection model working together and leading each other on our respective parts.

Challenges and Experimental results

The image processing of pose-detection is a very intensive task for CPU and is particularly suited for GPUs. In an Intel 7th Generation Quad-core processor can only analyze 0.5

frames per second for key-points using the MPII model. Our own system were limited to a very low frame rate despite powerful CPUs.

To overcome this limitation, we have created a virtual instance on Google Cloud Platform with a 64 vCPUs which can analyze achieves a 5 FPS. We have also set up a VNC server to output results of our cloud system and to test the results.

However, this process is significantly faster on GPU's. Therefore, to make our project more practical we decided to load the neural network on a Cloud GPU . However, running the code on cloud and setting up the code to run on a Graphics card with cuda support was very time consuming and difficult.

We have also tested different models such COCO model and hand gesture network for our code, which were significantly slower than the MPII pose detection Model, which lead us to choose the MPII model

Due to our limited time and access to the resources we were not able to load the neural network on a GPU and run image processing.

Conclusion

Due to being very computationally demanding the possibility of game controls through human motion is limited by using just CPU in a traditional computer. However, with models and better opencv support for running convolutional neural network models on GPU, computer vision can provide very interesting features for most simple games at relatively low cost. We have also successfully built our arcade game from scratch as a project and have built a UI and tuned the physics of the game with multiple user to make the game intuitive and fun while being challenging. In conclusion, the project brings two different aspect of arcade gaming and was able to demonstrate their potential when they are combined together.

Discussion on Future Improvements

There are a number of potential future improvement we would like to discuss in this section, which are the following,

1. Implementation of running the code in GPU: According to OpenPose github page, the openpose model can run 40 times faster on a GPU. This will allow video input of the player to be analyzed in approximately (40 x 0.5) = 20 FPS, which will definitely lead to a smoother and much rich gaming experience for the players.

2. Improvements to the visuals and graphics of the game: In the future we would like to add more professional looking sprites and background and implementation of an industrial UI while maintaining originality of the game. As both members of our team

were average at drawing in computers, it would be better if we got someone to professionally design some of the graphic element of the game.

3. With implementation of faster pose detection we would like to introduce multiplayer mode in split screen as the pose-detection algorithm is capable of detecting more than one pose without affecting it's run-time significantly. We can have upto 4 players in this way.

4. Though the process of creating the game we also learned a lot about programming design games After finishing the project there are a few things we would do differently in order to make the game more scalable and clean up the code. For example it would be nice to have a base game object that is extended by other game objects. This could be used in the engine for automatically calling the draw and update function of all game objects when needed instead of us adding them to the engine manually. This is not really necessary for this small game but as the number of game objects increased a management system for them would become necessary

Credits and References

Software/ Libraries:
SFML (zliv/png Licence)
ImGUI-SFML (MIT Licence)
Thor (MIT Licence)
ImGui (MIT Licence)
Opencv (3 Clause BSD Licence)
We also Used LearnOpencv As reference For getting the ML model working in C++ with opencv.

Sound clips used from :
https://freesound.org/people/E330/sounds/223082/ Sliding on snow sound
https://freesound.org/people/Breviceps/sounds/454595/ Crash sound
https://freesound.org/people/fschaeffer/sounds/337912/ Whoosh sound when flying

Background Music By: Ketsa
Licence: Creative Commons Attribution Non-Commercial
http://freemusicarchive.org/music/Ketsa/Different_Angles/Enticing_Dreams

Fonts:
Licence: Apache 2
https://www.fontsquirrel.com/fonts/permanent-marker

Sprites were used from :
Licence: https://pixabay.com/service/terms/#license (For Commercial Use No-Attribution)
https://pixabay.com/vectors/tree-trunk-nature-leaves-branches-576836/