

Flow Blade

Webcam-Based Optical Flow Interaction Prototype

By Ivo Nugent

Background

This project is Flow Blade, a fruit-slicing game that utilises a traditional webcam for play. The game itself is made in Processing, using OpenCV optical flow to detect movement in a player's camera view. The player is then tasked with moving their hand in real life in relation to the camera view shown to them, to slice fruit and avoid bombs. Each fruit has an arrow that indicates the required slice direction, which must be followed, or the fruit will not be cut, causing players to lose a life if they fall below the screen.

One of the main inspirations for Flow Blade was the popular phone game Fruit Ninja, which uses an arcade-style structure of slicing fruit by holding a finger on the screen while avoiding bombs. This game was primarily used as a design reference, inspiring the use of fruits and bombs in Flow Blade (Halfbrick Studios, 2010). Although Flow Blade draws on inspiring aspects of the game, it also uses two webcam-based games that operate similarly. These include a Runnable webcam project that tracks the player's hand and uses blob tracking, and an AI-controlled GitHub project that uses OpenCV, Pygame, and MediaPipe to make the hand act as a blade. (runnable, 2025; Verma, 2026). These examples illustrate the idea of using webcam interaction rather than typical touchscreens well. The main difference between the examples and Flow Blade is that slashing is managed entirely through OpenCV optical flow rather than using hand reference points. This means the prototype is less focused on specific hand tracking and more on whether movement is in the correct direction and at the correct speed for a clear slicing input.

The target audience for this project is quite broad, as the interaction itself is simple, with graphics and the game built with a focus on short arcade-style gameplay. The game includes three different modes: easy, medium and hard. These modes have different settings, as described in the section below. However, this change can make the game more accessible, as players can start with slower fruit and fewer bombs, whilst allowing more confident players to play at a faster, more intense pace. Optical Flow was the main driver for this project, as it is used to detect direction and speed from the camera, which helps keep the game feel meaningful in its interactions. This allowed the prototype to explore how a standard webcam can be used as an interaction device, highlighting its drawbacks and benefits without specialised, high-performance hardware.

Description and justification

User Interaction and Game Structure

The game follows a simple screen structure, so players can better understand it without being dropped into a 3, 2, 1 countdown, as was used previously. The main menu screen, as shown in Figure 1, introduces the game, the current high scores for each difficulty level, and the two modes. The player is shown a how-to-play page that can be accessed with the H key. From here, the player is given the game's basic rules and the other keybinds. Including the rules of slicing in the arrow direction, avoiding bombs, and not letting fruit fall. As shown in Figure two, the player is shown the key binds for the performance and vision modes, M for muting audio and ESC for leaving the game at any time. Having the how-to-play and main menu was necessary, as

webcam input and interaction are less normalised, so the player needs an explanation of what is going on before starting.

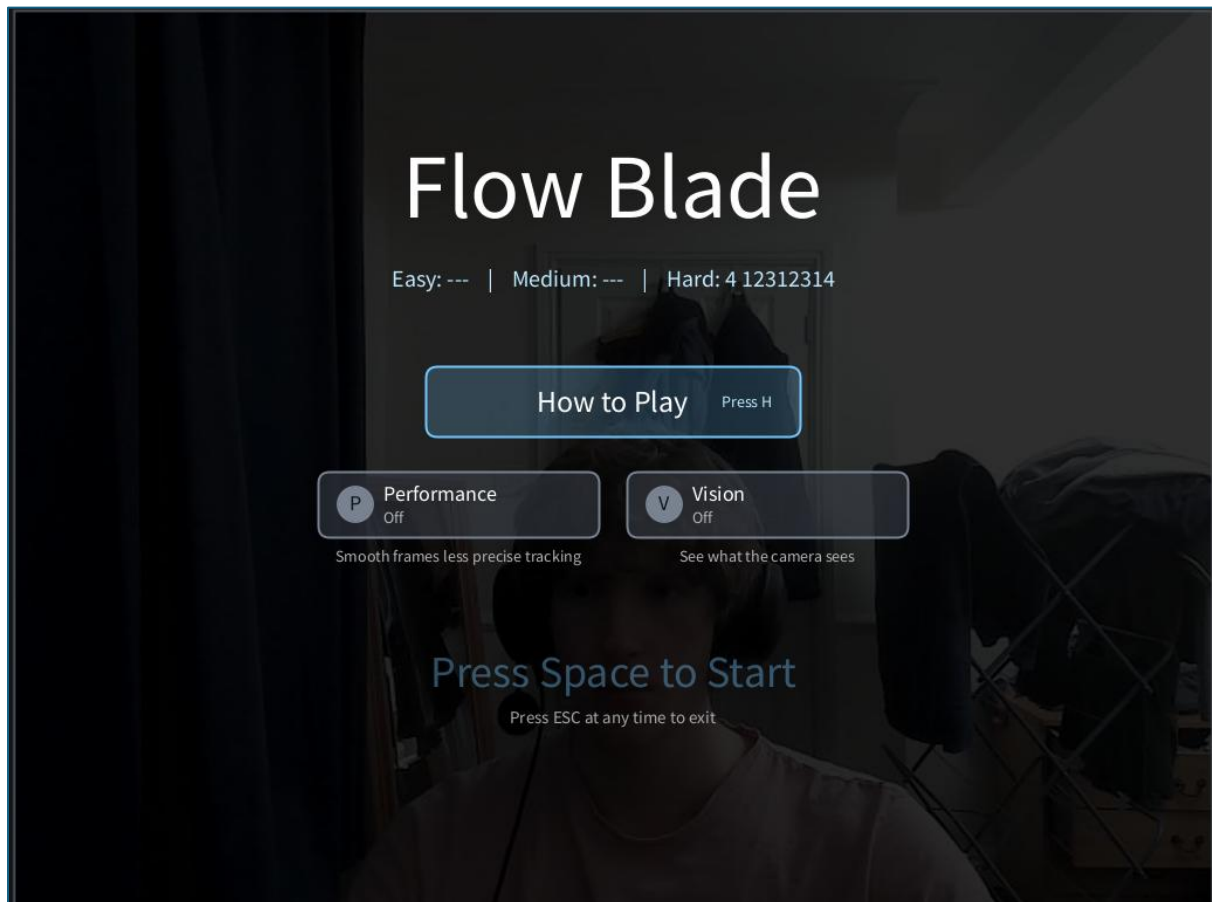


Figure 1 -Main Menu

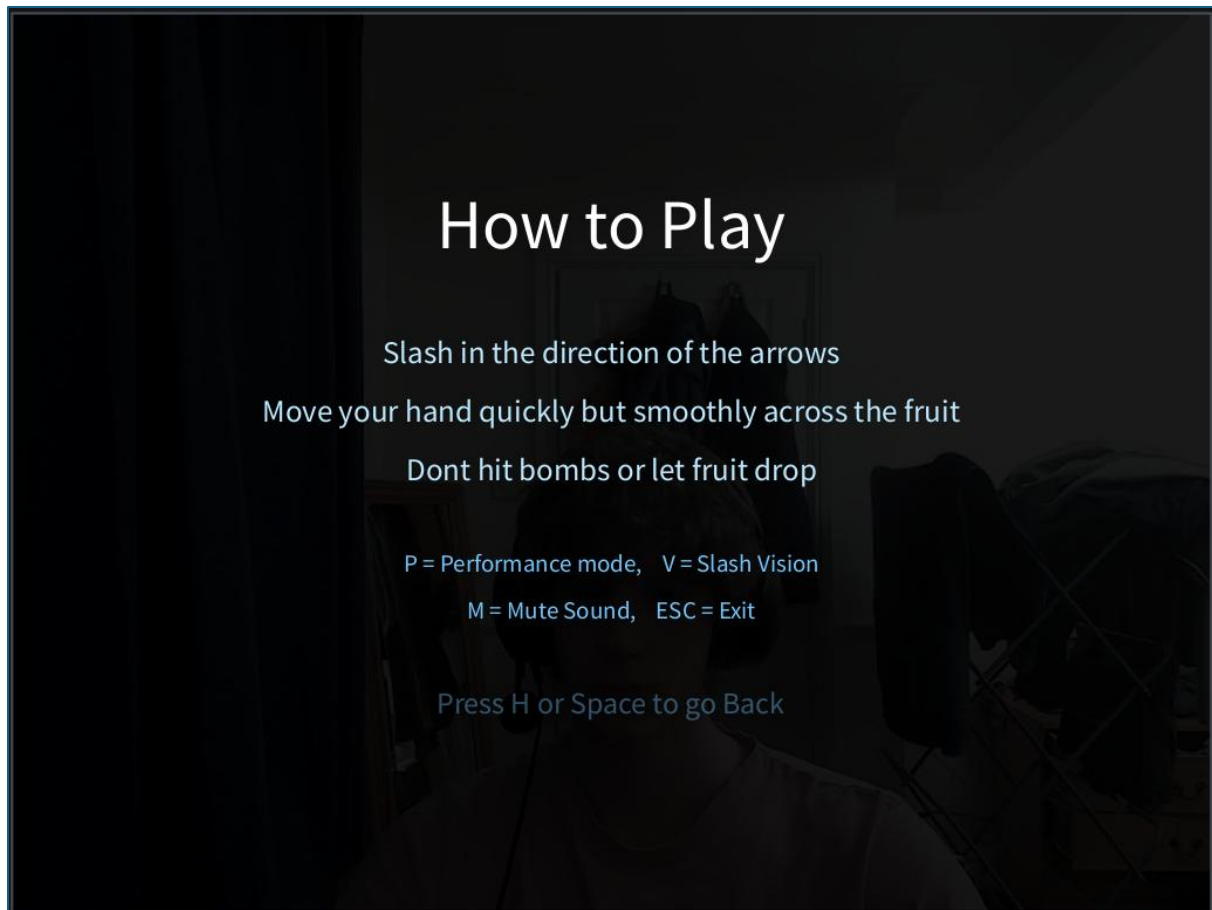


Figure 2 - How to Play

Once the player returns, they can press one of three keys to choose the difficulty: easy, medium, or hard. Once a difficulty is selected, a short 3-second countdown is provided to the user, giving them time to prepare and position themselves in front of the camera. During gameplay, the UI is simple, showing the score, lives, and occasional indicators of difficulty milestones and the need to slash faster. When all lives are lost, the player is moved to a death screen, then to an enter name screen, allowing a clear point before viewing the ending. This helps support the idea of replayability through score-based competition, as it incentivises people to achieve higher scores.

Vision-Based Interaction, Optical Flow, Slashing

Since the webcam is the main input device, the user must be given a clear display of the image to help the player understand the game and see what is going on more clearly. To aid this, the camera image is mirrored horizontally throughout the entire game. This makes the interaction feel more natural, since a player's right hand will appear on the right side of the screen. This was important because the player is attempting to aim physical movement to a screen, and having an unmirrored camera move feels highly confusing.

The webcam frames are processed using optical Flow from OpenCV to estimate the direction and magnitude of motion in each frame. In normal mode, this is updated every frame, whereas in performance mode, optical flow calculations are skipped every other frame, reusing cached grid to reduce computation. In the games context, a quick but smooth movement is treated as a slash.

Optical Flow fits well with the mechanics of slicing, as the game is entirely based on movement rather than recognising a hand. If the player moves slowly, the movement may be flagged as not good enough, with feedback such as "swing faster" provided. If the movement is good enough, the direction can be checked to ensure it is correct. This puts an interesting spin on the interaction, making it more skill-based rather than the two Fruit Ninja webcam examples, where waving your hand around slices everything. However, this Processing can be computationally demanding, and on weaker laptops, it can cause many issues. To avoid this, the camera frame is reduced to a smaller grid for optical Flow Calculation. The data is then stored in a grid, so the game checks the areas around the fruit and bombs rather than checking the entire screen for movement around all objects. This helps to keep responsibility whilst allowing for movement around all fruits and bombs to be detected. The tracked movement can also be displayed in vision mode, as shown in Figure 3, which helps illustrate how the system interprets and gauges player motion as a slash trail.

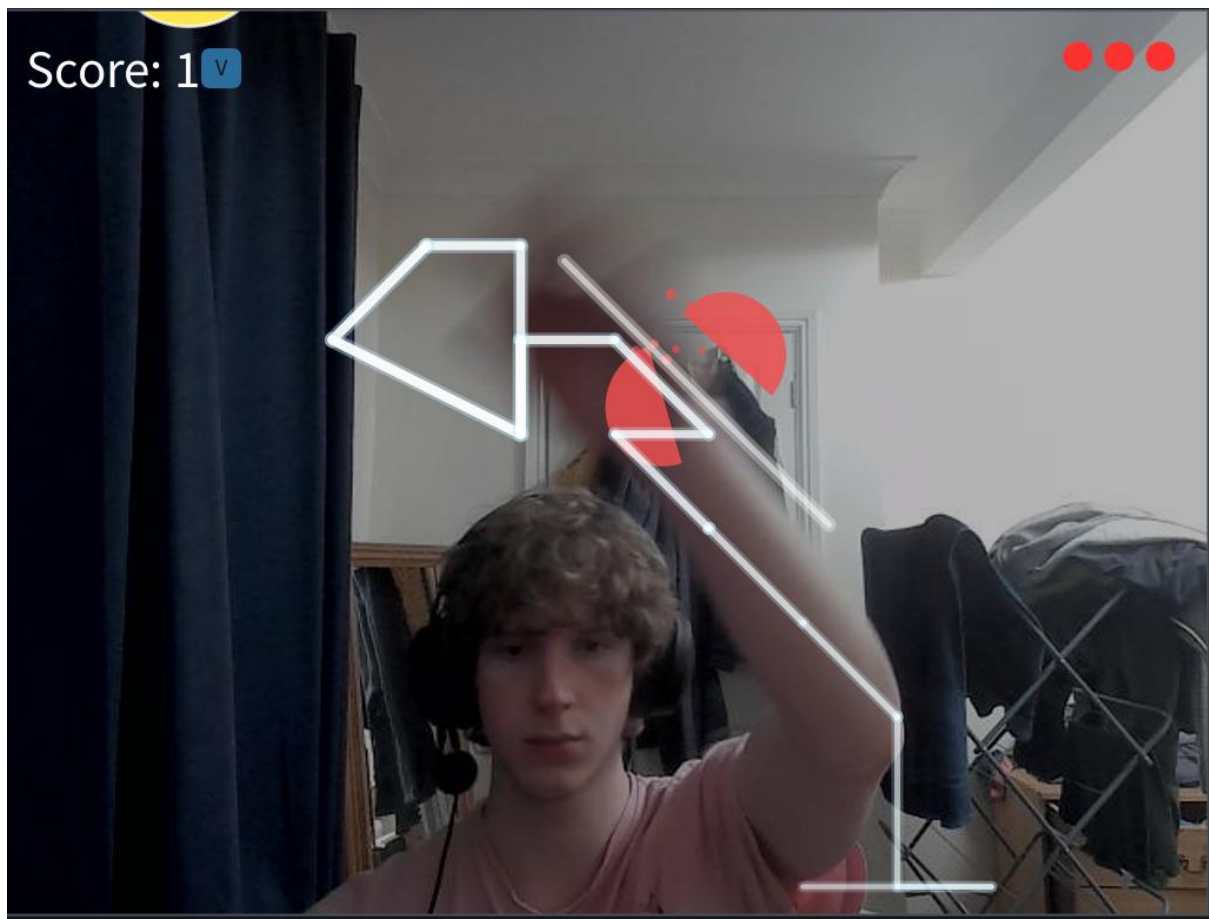


Figure 3 Vision Mode

Bombs, Feedback and Modes

Bombs are the main enemy and gameplay risk. Without them, the player could focus only on slicing fruit, but this would not incentivise challenge, precision, or slash management. When a bomb is struck, a player loses a life with a bomb sound and a reactive red screen shake. A flash arc is also used to make mistakes more noticeable, helping separate a bomb from a fruit in harder gameplay. The lives being set to three is a clear state of failure, where players understand

that losing all three means game over. This helps with the score and leaderboard, improving replayability.

Flow Blade features many instances of visual feedback, which are essential to the experience, as webcam interaction can feel stale compared to touch or mouse input. For instance, when a fruit is sliced, juice particles, fruit halves and a slice flash are shown as visual confirmation of a slash. The sound is used similarly to a bomb and slash sound on successful hits with either. During the game's development, a debug mode was utilised to check whether the slash trail was lining up and whether motion detection was working. This was useful for testing, so I turned it into a visual mode. Where users can toggle the same way as in debug mode to show the tracked slash trail, but in a cleaner, more visually appealing way. This can improve usability, as users can better understand the interaction during gameplay if they choose to use it.

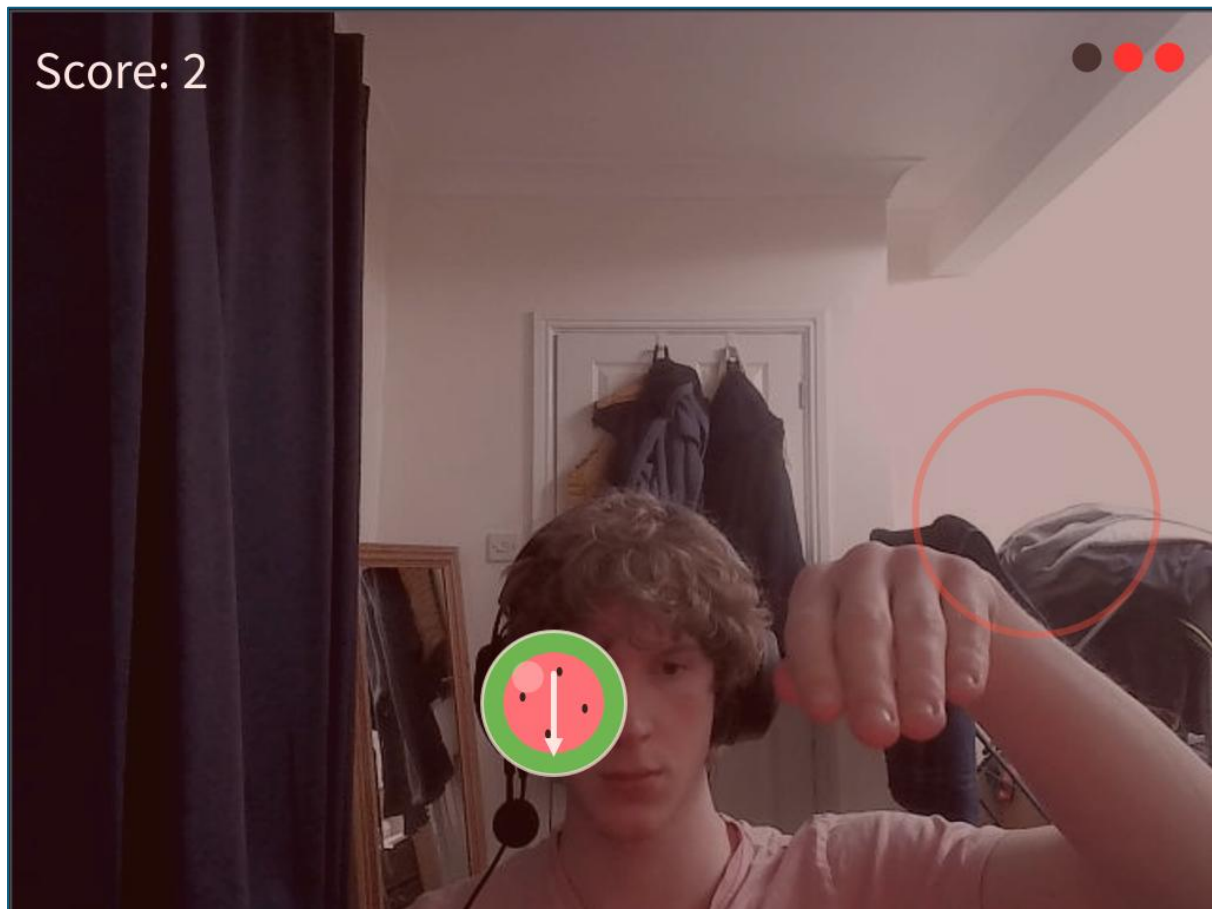


Figure 4 - Bomb Feedback

Difficulty and performance

Although briefly mentioned earlier, difficulty is very important in the game, as it offers more flexibility for new players. Easy mode has slower fruit, fewer bombs and less accuracy required on slashes. Medium provides a standard version of the game with slightly more fruit speed and bombs on easy mode. Whilst Hard has increased speed, frequent bombs, and accuracy that must be almost always on the white directional line. This helps make the game feel more demanding. Additionally, difficulty milestones are included in the game every 5 fruits successfully hit, where the speed or fruit count increases by 5 each time. This helps keep the core slashing interaction but changes how competitive the game is, rather than focusing primarily on unnecessary mechanics.

The more important mode is performance mode, due to the demanding OpenCV processing, especially on slower laptops. The performance mode helps to reduce computation through skipping optical-flow updates, reducing many visual effects and movement values. This allows the game to maintain the main interaction, even if it is running poorly.

Calibration was considered, but I chose not to implement it because I was unable to make it work as intended. As a result, the game uses fixed thresholds for motion detection; however, after testing, the values for easy, medium, and hard have been checked and deemed usable across different lighting conditions. Movement in the frame can still and will cause detection, which is unavoidable, as the game does not use specific hand tracking and is fully based on motion. This is not as accurate as a full calibration system, but it keeps the interaction still feeling intended.

Code Structure and AI use

The code is structured across 10 different Processing files to help keep it manageable. The main file handles setup, drawing, and states, while the game logic handles the active gameplay loop. The other files include the webcam and optical flow logic, as well as separate files for fruit, bombs, particles, audio, different screens, and the scoreboard. The webcam, as outlined earlier, was essential to be translated from unmirrored to mirrored. The lines with `translate(gameWidth, 0)` and `scale(-1, 1)` mirror the camera horizontally, making the interaction feel smoother and meaningful.

```
//draws mirrored webcam background
void drawCameraBackground() {
    rectMode(CORNER);
    noStroke();
    fill(0);
    rect(0, 0, gameWidth, gameHeight);

    pushMatrix();
    translate(gameWidth, 0);
    scale(-1, 1);

    int cameraTintAlpha = performanceMode ? performanceCameraTintAlpha : normalCameraTintAlpha;
    tint(255, cameraTintAlpha);
    if (webcamCapture.width > 0) {
        image(webcamCapture, 0, 0, gameWidth, gameHeight);
    }
    noTint();

    popMatrix();
}
```

Figure 5 - Mirrored Camera

As shown in Figures 5 and 6, the fruit arrows were among the most important parts of the experience. The arrow is drawn using the required `SlashDirection` vector, which is also used to check the player's slice direction. The end point of the arrow is calculated by multiplying the direction vector by a set distance from the fruit position. From this, a line is drawn across the fruit based on 8 possible directions. The arrow head is positioned using `translate`, then rotated based on the heading of `requiredSlashDirection`. This means that the arrow's visual direction is

based on the same value to help make the check of a successful slash easier. Gemini AI assisted with part of the rotation/vector maths, the function was adapted to my fruit class and connected to the slashes by myself.

```
void drawSliceArrow() {
    stroke(255);
    strokeWeight(5);

    PVector pointerEnd = new PVector(
        xPosition + requiredSlashDirection.x * 30,
        yPosition + requiredSlashDirection.y * 30
    );

    line(
        xPosition - requiredSlashDirection.x * 25,
        yPosition - requiredSlashDirection.y * 25,
        pointerEnd.x,
        pointerEnd.y
    );

    pushMatrix();
    translate(pointerEnd.x, pointerEnd.y);
    // Gemini AI code
    rotate(requiredSlashDirection.heading());
    //
    fill(255);
    noStroke();
    triangle(0, 8, 0, -8, 16, 0);
    popMatrix();
}

//checks if fruit was missed
boolean isOffScreen() {
    return yPosition > gameHeight + fruitSize;
}
}
```

Figure 6 - Arrow Slicing

```
PVector[] possibleDirections = {
    new PVector(1, 0),
    new PVector(-1, 0),
    new PVector(0, 1),
    new PVector(0, -1),
    new PVector(1, 1).normalize(),
    new PVector(-1, -1).normalize(),
    new PVector(1, -1).normalize(),
    new PVector(-1, 1).normalize()
};
```

Figure 7 - Possible Directions

Figure 7 shows the cached grid for optical Flow. This was the more complex part of the implementation and was assisted by Generative AI. This code is an updated version of a previous one that tracked every fruit and bomb; instead, the screen is divided into cells that

store optical values. When the bombs or fruit need to be checked for movement, the game averages the movement of the sampled cells around the object. This implementation improved overall performance, as the game can reuse cached motion rather than performing repeated calculations for all objects.

```
PVector sampleCachedFlow(float regionX, float regionY, float regionWidth, float regionHeight) {
    if (!flowGridReady) {
        return new PVector(0, 0);
    }

    int startColumn = constrain(int(regionX / flowGridCellSize), 0, flowGridColumns - 1);
    int endColumn = constrain(int((regionX + regionWidth) / flowGridCellSize), 0, flowGridColumns - 1);

    int startGridRow = constrain(int(regionY / flowGridCellSize), 0, flowGridRows - 1);
    int endGridRow = constrain(int((regionY + regionHeight) / flowGridCellSize), 0, flowGridRows - 1);

    PVector combinedFlow = new PVector(0, 0);
    int sampledCellCount = 0;

    for (int gridColumn = startColumn; gridColumn <= endColumn; gridColumn++) {
        for (int gridRow = startGridRow; gridRow <= endGridRow; gridRow++) {
            combinedFlow.add(cachedOpticalFlowGrid[gridColumn][gridRow]);
            sampledCellCount++;
        }
    }

    if (sampledCellCount > 0) {
        combinedFlow.div(sampledCellCount);
    }

    return combinedFlow;
}
```

Figure 8 Sampled Cache flow

The slice effect in the game uses a slash direction to draw an animated line across the fruit. The start and target points are based on the direction vector, utilising lerp to animate the line. This was essential, as feedback needed to be provided to players when a fruit is sliced, so they could see which camera input or speed caused a successful slash and replicate it.

```

//draws effect on screen
void drawToScreen() {
    float lineLength = flashSize * 1.5f;

    float startingX = xPosition - slashDirection.x * lineLength;
    float startingY = yPosition - slashDirection.y * lineLength;

    float targetPointX = xPosition + slashDirection.x * lineLength;
    float targetPointY = yPosition + slashDirection.y * lineLength;

    float animatedX = lerp(startingX, targetPointX, animationProgress);
    float animatedY = lerp(startingY, targetPointY, animationProgress);

    stroke(255, remainingOpacity * 0.6f);
    strokeWeight(12);
    line(startingX, startingY, animatedX, animatedY);

    stroke(255, remainingOpacity);
    strokeWeight(5);
    line(startingX, startingY, animatedX, animatedY);
}

//checks if effect finished
boolean isDead() {
    return remainingOpacity <= 0;
}
}

```

Figure 9 - Draw Slash

Generative AI Use

Generative AI from both ChatGPT and Gemini AI was used throughout this project. The main use was for specific technical maths-related bugs and issues. Some functions were fully AI-generated, including cached optical-flow grid, refresh functions, high-score management and leaderboard loading. AI was used to assist with complex maths, mapping optical Flow to screen regions, sampling from the cached Flow, checking whether the hidden slash trail hit a fruit, handling it, and splitting/particle movement. All generative AI usage in the code has been documented in code comments. All AI code, regardless of assistance or full generation, was checked and tested to ensure that any suggestions made fit the project.

Reflection

Overall, the Flow blade has been covered, and a prototype of a vision-based interaction game has been successfully created. Many aspects of the game contributed to its success, where webcam input very closely matches the slicing action as players move their hands to create a slash without the need for a mouse or keyboard. The mirrored camera background and vision mode help make the interaction easier to understand. Skill-based incentives are strong, with directional arrows used in conjunction with difficulty milestones, making scores above 25 very impressive, as slashes need to be fast and precise. This links to one of the first limitations, as using two hands at the same time does not work. This may be considered a flaw, as the challenge may incentivise players to use two hands when it becomes difficult. Additionally, the use of direction helps avoid the issues I noticed in all three referenced games, where any slash goes with no direction or methodology for play.

The technical implementation was successful yet limited. Optical Flow was extremely suitable for the project's scope, as movement is detected well and can be based on direction and strength rather than a hand shape. This made the interaction simpler to implement whilst still allowing the game to check whether movement was quick enough and close enough to a fruit or a bomb. The use of a cached flow grid, in addition to the performance mode, helped the game run much better, as significant lag was experienced during development. This, however, poses the main limitation of flow-based detection: it detects movement rather than a specific hand. This means the largest flaw is how background movement, lighting, camera quality, or any movement from unintended parts of the body could cause issues and unreliability. This also links back to the fact that using two hands is not plausible, as the system is not able to differentiate between them, causing bugs where stronger movement takes priority. Whilst this does not cause game-breaking issues, it is definitely a consideration, as it limits overall interaction precision, which could cause player frustration if too much movement is occurring in the background. The second-largest limitation was the use of calibration; the intended original use was to have the player swipe three times and use average to allow for more realistic gameplay. This, however, would not work as intended due to slowness, detection issues, and increased code complexity. This was decided not to be implemented, and fixed thresholds were used instead. This kept the game feeling accurate during threshold testing without the need for calibration.

Generative AI provided significant value in the project's development, particularly for complex maths and the use of cached optical flow, among other code issues. AI generated entirely some of the code used, whilst some was simple suggestions or assistance with small issues or coding structure. Some AI use could have been avoided; in some cases, when it suggested what to do, I believe I could have reached the same outcome if given enough time. However, the use of AI here was to save time on potential hours of work that could have been spent resolving issues. If I were to continue project development, I would likely focus on implementing calibration, given the time and complexity constraints; I was not able to implement it. Additions fall under both technical and design, where improved scoring with combos inspired by Fruit Ninja, or special

power-ups, say, if the user were to move their hand in a circular motion, could have been interesting. However, the success of the optical flow detection makes future development feel as though it should be kept as the main advanced interaction.

Word Count: (2722)

Bibliography

Halfbrick Studios (2010) *Fruit Ninja Classic*. Available at:

<https://www.halfbrick.com/games/fruit-ninja-classic> (Accessed: 21 May 2026).

runable (2025) *Built a webcam Fruit Ninja where you slash with your actual hand*. Reddit. Available at:

https://www.reddit.com/r/vibecoding/comments/1psw178/title_built_a_webcam_fruit_ninja_w_here_you_slash/ (Accessed: 21 May 2026).

Verma, N. (2026) *Fruit Ninja - AI Hand Controlled*. GitHub. Available at:

https://github.com/devnickverma/Fruite_Ninja_Using_OpenCV (Accessed: 21 May 2026).

Appendices:

Appendix A: Acknowledgement's

Grammarly Writing Assistant. Accessed: 21/05/26 Available at: (<https://app.grammarly.com>)

Use case: Grammarly Review Suggestions to improve grammar, clarity and sentence structure.