Style ***T***racking ***E***xpressive ***P***ad ***S***ystem

Arcade Rhythm Game with Pose Detection

Group 8 Authors

Christopher Solanilla	Jani Jon Lumibao	Kaila Peeples	Andres Abrams	Blake Whitaker
Computer Engineering	*Computer Engineering*	*Photonics and Science Engineering*	*Computer Engineering*	*Electrical Engineering*

Reviewer Committee:

Dr. Stephen Eikenberry — CREOL — Professor
Dr. Saleem Sahawneh — ECE — Lecturer
Dr. Justin Phelps — ECE — Professor

Mentor:

Dr. Chung Yong Chan
Dr. Aravinda Kar

Table of Contents

List of Tables

List of Figures

1. Executive Summary

The landscape of arcade rhythm games has traditionally focused on a singular metric of success: timing accuracy. This paradigm often overlooks a key element of dance: expressive performance. The Style Tracking Expressive Pad System (STEPS) project was conceived to address this gap by creating a new generation of dance games that rewards both precision and creativity. This project delivers a fully integrated arcade-style system, from custom hardware to a specialized game engine, centered around an innovative computer vision system that quantitatively scores a player's stylistic movement.

The core of the STEPS project is a custom 9-panel dance pad that merges the gameplay styles of iconic rhythm games into a unique, hybrid experience. To meet the demanding requirements for reliability and complexity, a sophisticated modular hardware architecture was developed. This distributed system is composed of three distinct custom Printed Circuit Boards (PCBs): a central Master Controller, a high-current Power Hub, and nine identical "smart" Tile Boards. Each tile is an independent, microcontroller-driven module responsible for its own sensor reading and aesthetic LED feedback, all communicating with the Master Controller over a robust I2C bus. This advanced, modular design creates a scalable and easily maintainable hardware platform.

The primary innovation of the STEPS project is its real-time, vision-based "Style Score" system. This system utilizes a camera and an 850nm infrared (IR) LED illumination ring, driven by a custom constant-current driver, to ensure reliable player tracking in any lighting condition. Using the MediaPipe pose estimation framework, the system analyzes the player's full-body movements and detects when they successfully execute predefined, expressive poses during gameplay. This adds a new dimension to the genre, encouraging players to engage in dynamic, full-body performance rather than just minimalistic, efficient footwork.

The successful integration of these three pillars: custom modular hardware, a tailored software game engine, and a real-time computer vision system, represents a significant engineering achievement. The STEPS project successfully pushes the boundaries of interactive entertainment, delivering a proof-of-concept for a new style of rhythm game that is more engaging, physically expressive, and technically advanced than its predecessors. The final prototype is a testament our team's multidisciplinary skills in embedded systems design, power electronics, software development, and optical engineering, laying the groundwork for a potential commercial product for both arcade and home use

2. Project Description

2.1 Motivation and Background

In recent years, rhythm games have surged in popularity among both casual and competitive gamers. Titles like *Dance Dance Revolution (DDR)*, *Pump It Up (PIU)*, and *StepManiaX* offer not only fast-paced gameplay but also unique forms of physical interaction that make them stand out from traditional video games. For readers unfamiliar with rhythm games: players typically choose a song, and as the music plays, visual cues (usually arrows) scroll on the screen toward a target zone. Players must step on corresponding arrows on the dance pad in time with the music. Successful timing earns points, while misses break combos and reduce scores. In the games aforementioned, players are often using their feet to step on panels that are labeled with directional arrows. In DDR it is with the up down left right arrows, StepManiaX has the same but with an additional center arrow, and Pump it Up has diagonal arrows instead of the 4 up down left right arrows with the center note as well.

Figure 2.1 *StepManiax on UCF at Knightros*

As lifelong fans of rhythm games, we've always aspired to create a game that blends the fast-paced footwork of DDR and PIU with innovative mechanics that reward not only timing accuracy but also expressive performance. This passion became the foundation for our senior design project: a custom-built rhythm arcade machine that reimagines the traditional dance pad format. Our system features a unique 9-panel layout, combining the four cardinal directions, four diagonals, and a center panel. This effectively merges the core mechanics of DDR and PIU into a new hybrid experience.

To take it a *step* further, we are integrating a computer vision system capable of analyzing player movement during gameplay. This system detects whether the player completes the charts with minimal effort or performs dynamic, stylish movements such as spins or arm gestures. Players who demonstrate expressive flair are rewarded through a secondary metric we call the Style Score, adding a new dimension to gameplay that celebrates both precision and creativity.

Our team brings a diverse range of skills and experiences to the project. Our Computer Engineering majors contribute a strong passion for rhythm games, hands-on experience from developing a basic dance pad prototype in the past, and solid experience in hardware integration and software development. One of our teammates has a background in dance, offering valuable insight into expressive movement, physical design, and is also well-versed in PCB design. Another teammate serves as the president of the UCF Esports Club, providing a competitive gaming perspective that helps shape our gameplay mechanics and balance. Our team is rounded out by a Photonics Science and Engineering student who brings specialized expertise in optics for our computer vision system, and an Electrical Engineering student who serves as our hardware lead.

We believe this blend of technical, creative, and performance-oriented backgrounds makes our team uniquely positioned to create a rhythm game that is both entertaining and technically ambitious. By combining game development, embedded hardware, and real-time computer vision, we aim to push the boundaries of traditional rhythm games. Ultimately, we envision this system not just as a school project, but as a potential commercial product suitable for both arcade and home use. Overall, our project is a tribute to the genre we love and an innovative leap forward in how rhythm games are played.

2.2 Existing Product/Past Project/Prior Related Work

2.2.1 Dance Dance Revolution

Dance Dance Revolution (DDR), developed by Konami in 1998, is one of the most iconic rhythm games in the world. Players step on a 4-panel dance pad — with up, down, left, and right directional arrows — in time with scrolling on-screen cues synchronized to music. DDR is widely recognized for its role in popularizing rhythm games globally and has been featured in both arcade and home console formats. It uses pressure-sensitive panels and a scoring system based on timing accuracy, rewarding "Perfect," "Great," or "Miss" for each input. However, the game focuses purely on foot-based precision and lacks a scoring component for stylistic or expressive movement. Players who still choose to complete charts with style are called “free stylers” and are highly respected in the community for completing easy to mid level charts with very hard expressive and complicated movement. This movement can range anywhere from spinning, to swaying the arms, to handstands and break dancing.

2.2.2 Pump It Up

Pump It Up (PIU), developed by Andamiro in 1999, is a 5-panel dance rhythm game that includes four diagonally placed panels and a center panel. PIU emphasizes freestyle movement more than DDR and is especially popular in South Korea. While PIU retains the same timing-based scoring mechanics as DDR, it introduces more physically varied and complex choreography due to its diagonal input layout. Nonetheless, PIU still lacks any integrated camera system or style-based scoring . Just like DDR, PIU has the same respected players who complete charts with freestyling despite still not being rewarded for doing so in game.

2.2.3 Dance Around

Dance Around is a rhythm game similar to Dance Dance Revolution but instead of using pressure-sensitive dance pads, it relies solely on a camera-based motion tracking system. This is done by using VisionPose to generate a 3D model of the player’s body and assessing their dance performance based on their ability to match the poses given and their own expressive movements[2]. During the game, players are prompted to mimic target poses displayed on the monitor, with visual cues that indicate the proper hand or foot placement. However, customer feedback has highlighted several limitations of this system. Customers have stated that the game has consitently misevaluated full body movements and is only capable of capturing the hand and foot positions of the players reliably[3]. Additionally, customers have noted that the calibration process could be overly lengthy, which detracts from the ease of use.

2.2.4 Dancerush Stardom

Developed by Konami, DANCERUSH STARDOM is a freestyle rhythm dance game that eliminates the traditional dance pad structure in favor of a large pressure-sensitive surface. The game uses a camera system to provide feedback and record gameplay, but the scoring is still based on foot movement across a large flat pad with visual indicators for steps and slides. The game promotes freestyle dancing, including spins and slides, and is considered more modern and expressive than DDR. However, like Dance Around, it does not feature a true pose recognition or performance grading system. Flair and dance expression are encouraged but not quantitatively rewarded within the game mechanics.

2.2.5 StepManiaX

StepManiaX is a rhythm game inspired by DDR and PIU and developed by the creators of StepMania. It uses a 5-panel pad layout with center, up, down, left, and right panels, and is designed for high durability and fitness applications. StepManiaX includes a touchscreen interface and modernized music selection UI, with content designed to be accessible for both casual and serious players. While it modernizes the user experience and supports a wide range of difficulty levels, it retains the traditional scoring focus on timing accuracy. There is no implementation of camera-based tracking or any scoring system that rewards visual expression or dance style beyond note timing.

Figure 2.2 *Illustration prototype of StepManiaX, a 5 panel Dance Rhythm Game Arcade Cabinet[6]*

2.2.6 Microsoft Kinect

Computer vision (CV) technologies have seen increasing integration into modern games, from facial detection and hand tracking to body motion recognition. These advancements have allowed developers to create more immersive and interactive gaming experiences by enabling players to interact with virtual worlds using natural body movements rather than traditional input devices.

The Microsoft Kinect, released in 2010 for the Xbox 360, represented a major leap forward. Utilizing an infrared depth sensor and RGB camera, the Kinect could track full-body skeletons in real-time using a technique called skeleton tracking, which mapped 20 joints of the player's body. This enabled games like *Kinect Adventures* and *Dance Central* to support full-body gameplay without the need for controllers. The Kinect SDK and the associated machine learning models enabled gesture detection and activity recognition, expanding the use of CV beyond entertainment into healthcare, education, and robotics. However, challenges such as latency, lighting interference, and limited tracking accuracy in multi-person scenes persisted.

While the Kinect was not utilizing conventional Computer Vision techniques and opted for IR sensing, it was able to prove very effective for the time allowing less powerful hardware like the xbox to perform simple calculations. By analyzing distortions in an infrared dot pattern, Kinect could calculate depth and track motion without any physical markers.

To make skeletal tracking accessible to developers, Microsoft released the Kinect SDK, enabling applications to extract joint data and recognize poses. A foundational feature of this SDK was its ability to detect up to 20 distinct skeletal joints, as visualized in Figure 2.3. These include major points such as the head, spine, hips, shoulders, and limbs. This allowed developers to build gesture-based interfaces, games, and virtual assistants capable of interpreting full-body movement.

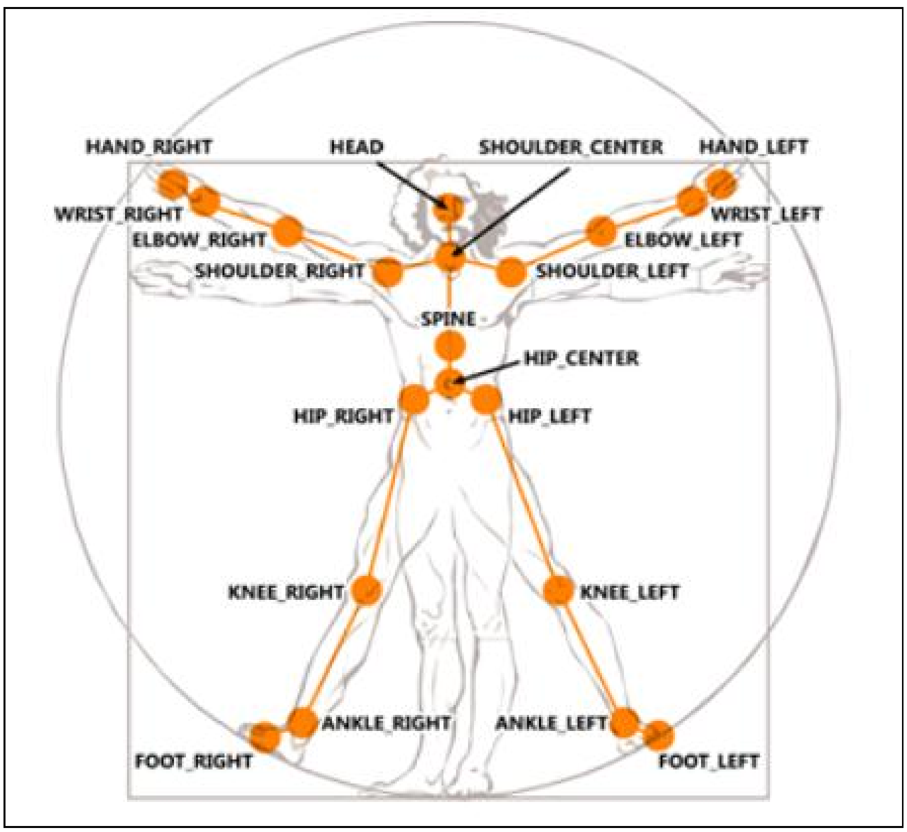


Figure 2.3 *Joint layout used by Microsoft Kinect's skeletal tracking system*

Kinect's architecture proved revolutionary for its time. The use of the various hardware shown in Figure 2.3 such as the IR Emitter, Color Sensor, IR Depth Sensor, and Tilt Motor allowed the Kinect to capture accurate per-pixel values along with depth measurements even in low light shown in 3.3.2.C. These technologies were critical in early gesture-based games like *Dance Central* and inspired later research in human-computer interaction.

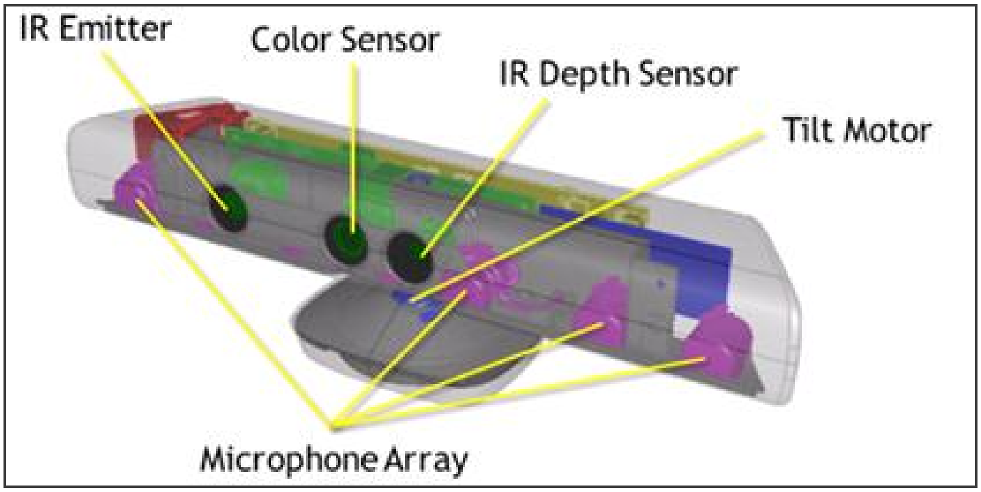


Figure 2.4 *Kinect Architecture*

Figure 2.5 *3D Depth Image in Grayscale*

Microsoft also provided developer tools like Kinect Studio, which recorded and replayed interaction data to aid debugging and testing. Applications could be built rapidly in WPF using templates like KinectSeletonApplication which visualized tracked joints as ellipses on a canvas. The Kinect SDK also introduced features like seated mode, facial tracking, and support for close-range users via *Near mode*, allowing for more flexibility in design.

Kinect's legacy lives on in modern computer vision applications. Its model of joint-based tracking and use of multimodal input (pose + voice) laid the groundwork for today’s AI-powered body tracking with neural networks and webcams. While contemporary systems like MediaPipe Pose and OpenPose no longer require specialized hardware, the principles they followed, path estimation, joint modeling, and user-centric design, remain deeply rooted in Kinect's innovations.

2.2.7 Just Dance

Another notable application of CV in gaming is Just Dance, a franchise by Ubisoft launched in 2009. Originally designed for the Nintendo Wii using motion controllers, newer iterations supported Kinect and PlayStation Camera input. Just Dance evaluates player movements by comparing their captured silhouette or skeleton data to pre-recorded dance routines. While this system simplifies feedback into generalized “good” or “perfect” scores, it creates an accessible experience that emphasizes fun over precision. This trade-off reflects a key challenge in CV-based gameplay: achieving a balance between technical accuracy and player enjoyment.

2.2.8 Pokemon Go

In recent years, computer vision has enabled augmented reality (AR) games such as *Pokémon GO*, which use smartphone cameras and visual SLAM (simultaneous localization and mapping) to blend virtual content with the real world. Although AR relies more heavily on scene understanding than pose detection, these games showcase how CV can extend interactivity beyond fixed hardware setups.

2.2.9 Mediapipe

More recently, open-source tools like MediaPipe have democratized access to pose estimation algorithms that were once exclusive to large tech companies. With these libraries, developers can extract real-time body, hand, and facial landmarks using only a standard webcam. This has fueled a resurgence in indie games and research projects that explore motion-based gameplay and accessibility.

2.2.10 Chunithm

Similarly, Chunithm, a rhythm game by Sega, relies on advanced hand-tracking through infrared or camera-based systems to detect fast, complex gestures like air slashes or upward swipes. While not all the technical details of these proprietary systems are public, they demonstrate how CV, including neural network-based recognition and 3D hand pose estimation, has become viable even in high-speed arcade settings.

2.3 Project Goals

The main goal of this project is to design and build a self-contained arcade-style rhythm game system that includes both the physical dance pad hardware and the video game software. The system will feature a 9-panel layout to allow for expanded gameplay mechanics. In addition to foot-based input, the game will include a computer vision system that evaluates the player's expressive movement using a live camera feed. Our goal is to create an experience that rewards both precision and performance, allowing players to interact with the game through both steps and body movements. The final product will include a working rhythm game engine, a functioning pad-to-PC interface, custom chart creation tools, and vision-based style scoring. While the game will initially run on a standard PC or laptop, we are also exploring the feasibility of deploying the full system on an embedded platform such as the NVIDIA Jetson Nano. This would allow for a compact, standalone arcade unit, and is being considered as a stretch goal pending performance testing and thermal optimization.

2.3.1 Hardware Goals

- **Basic:**
    - develop 9-direction interactive arrow pads with RGB LED feedback for each direction by the end of SD1
    - ensure responsive and accurate force detection using force-sensing resistors (FSRs) with an input response time of 5-10 ms
    - design and integrate a 3-4 mm aspherical lens that maximizes image sharpness (≥ 3 pixels/mm) across a 2.9 m x 2.9 m field of view while taking the depth of field (DOF) into account.
    - implement a synchronized time-multiplexed near-infrared (NIR) LED illumination system with ≤17 ms full-zone cycle to reduce power usage and prevent flicker.

- **Advanced:**
    - reduce total PCB computational processing time by at least 30% compared to SD1 prototype using firmware optimizations.
    - re-design pad layout to reduce total surface area by at least 15% compared to the SD1 prototype, by SD2 midterm demo, to enhance portability without sacrificing gameplay accuracy
    - select and implement LED beam angles and placement configurations that achieve ≥85% measured brightness uniformity with minimal shadowing across all four zones during optical subsystem testing in SD2.

- **Stretch:**
    - integrate Bluetooth or Wi-Fi module to enable wireless data transmission between pads and central controller
    - create a foldable dance pad prototype with < 3cm thickness when collapsed, suitable for storage or transport

- reduce input latency to 1-5 ms through optimized ADC sampling and FSR circuit tuning.
- run the entire software stack on an NVIDIA Jetson Nano, including vision, scoring, and display modules

2.3.2 Software Goals

- **Basic:**
  - develop a rhythm game engine tailored to the 9-panel pad layout with frame rates ≥ 60 FPS
  - implement a user interface for local song selection and post-game performance feedback
  - run MediaPipe-based pose estimation locally on the system (e.g., Jetson nano or PC)
  - Implement pose-based scoring by triggering bonus effects for matching target poses during gameplay

- **Advanced:**
  - include a song editor feature that allow users to generate custom choreography charts
  - load a library of at least 6 preloaded songs at launch for player testing
  - improve pose detection robustness with optimized frame filtering and landmark smoothing (e.g., 95% detection confidence)

- **Stretch:**
  - add player login with global leaderboard integration via secure backend
  - launch a mobile app that syncs gameplay statistics, average scores, and playtime
  - add configurable startup modes (Arcade/Home) with different default UI settings and gameplay speeds

2.4 Project Objectives

To guide the development of the STEPS system, we have defined a set of core objectives that encompass the design, fabrication, and integration of the project's primary hardware and software subsystems.

- **Hardware Fabrication:**
  Construct a durable 9-panel dance pad using custom-designed Printed Circuit Boards (PCBs) and Force-Sensing Resistors (FSRs). The system will be engineered to achieve a low-latency input response time of less than 10ms and communicate with a host PC as a standard USB HID device. Each panel will be pressure-sensitive and capable of reliable input across a wide range of user weights and step patterns. Additional hardware considerations include robust

electrical connections, compact sensor mounting, and modularity to allow for maintenance or upgrades.

- **Game Engine Development:**
  Build a complete rhythm game from the ground up in the Godot engine. This software will feature a user interface for song selection and score display, a system to load custom song charts, and immediate audio-visual feedback for player accuracy. The game will be designed to accommodate various skill levels and allow user-generated content to be imported easily. The engine will support real-time synchronization between the music, step inputs, and lighting cues to maximize immersion

- **Optical System Design:**
  Engineer an integrated imaging system for reliable player tracking in various ambient lighting conditions. This involves selecting an appropriate camera sensor and designing a custom aspherical lens system, supported by a time-multiplexed 850nm infrared (NIR) LED illumination array. The optical design will focus on achieving consistent visibility of the player's body regardless of external lighting or room setup, ensuring that the pose estimation system operates with high fidelity in dynamic play environments.

- **Vision-Based Scoring:**
  Implement a real-time "Style Score" system using a pose estimation library like MediaPipe to extract a player's full-body keypoints. This system will detect a series of predefined expressive poses and award bonus points to reward player creativity beyond simple timing accuracy. The vision module will operate in tandem with the core game logic to ensure low-latency detection and minimize false positives. Future work will explore machine learning-based classification to expand the range of recognized poses and improve robustness.

- **Full System Integration:**
  Integrate all hardware and software components into a cohesive and functional prototype. This final objective involves verifying that the physical dance pad seamlessly controls the game engine and that the pose detection system provides accurate input to the scoring logic, delivering a stable and engaging user experience ready for demonstration. A core emphasis will be placed on minimizing system latency, ensuring modular connections between components, and supporting long-duration playtesting to evaluate durability and feedback performance.

2.4.1 Prototype illustration/Blueprint

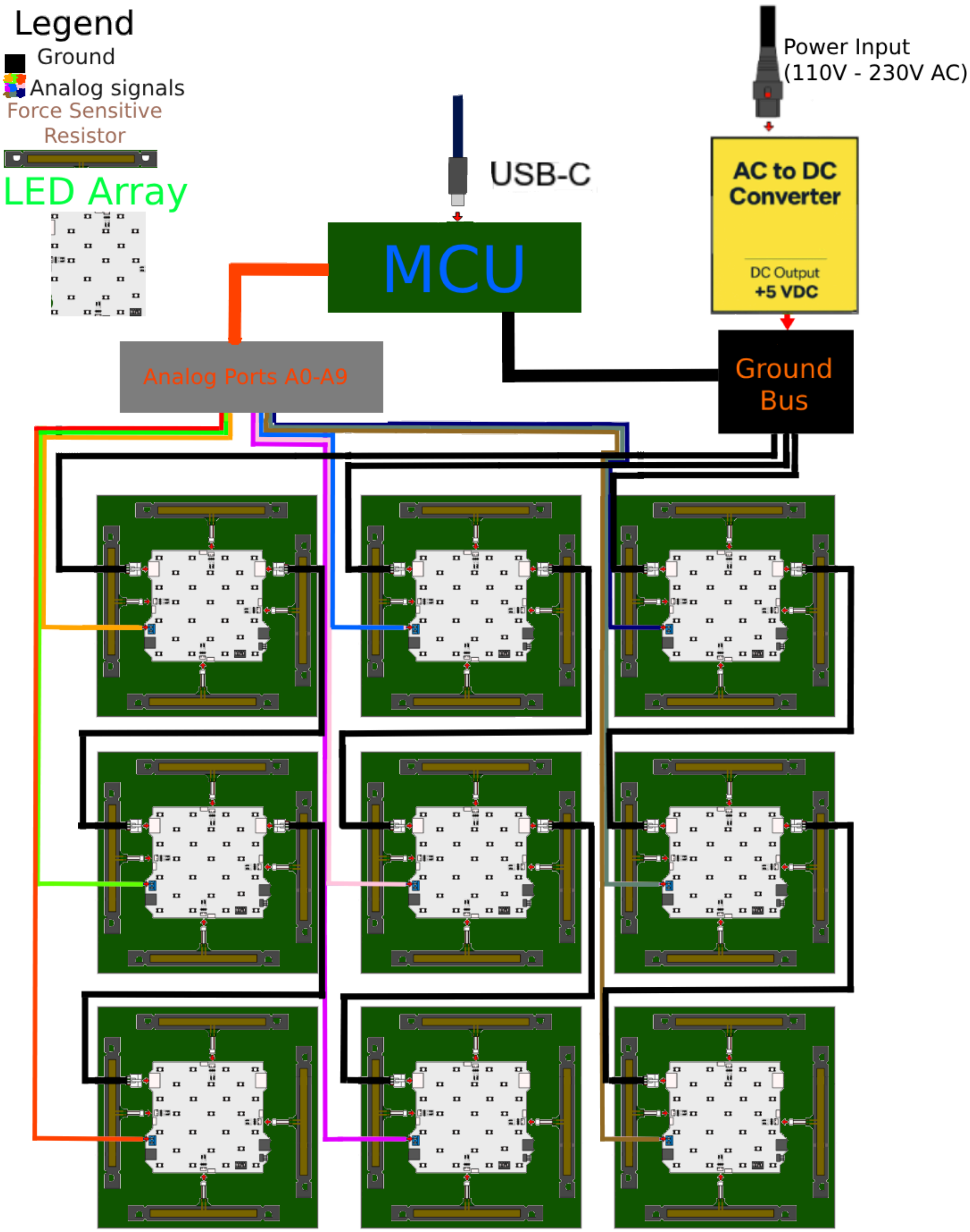


Figure 2.6 *Electronic Blueprint Prototype Illustration*

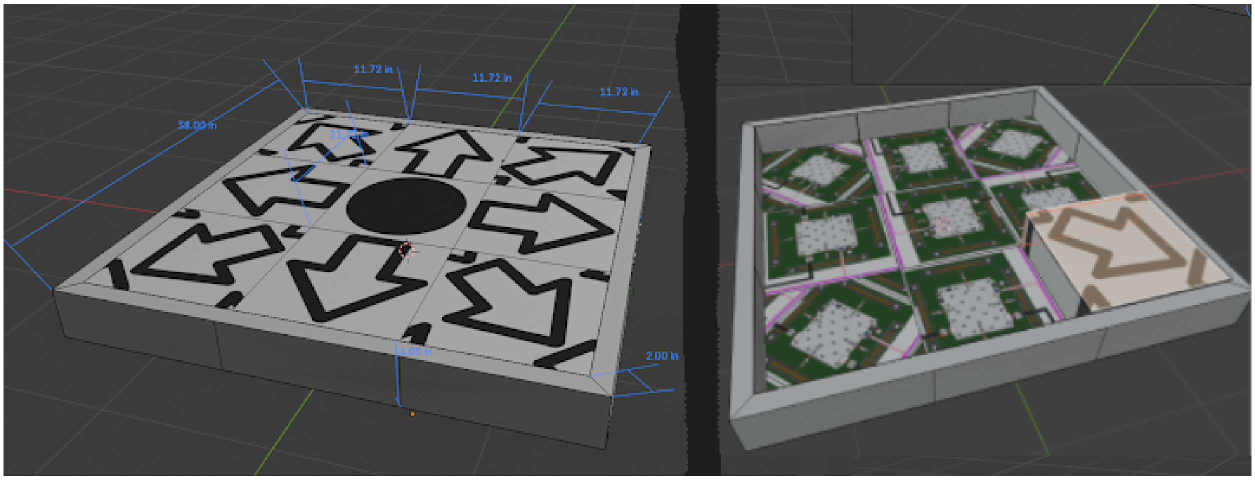


Figure 2.7 *Dance Pad 3D Model*

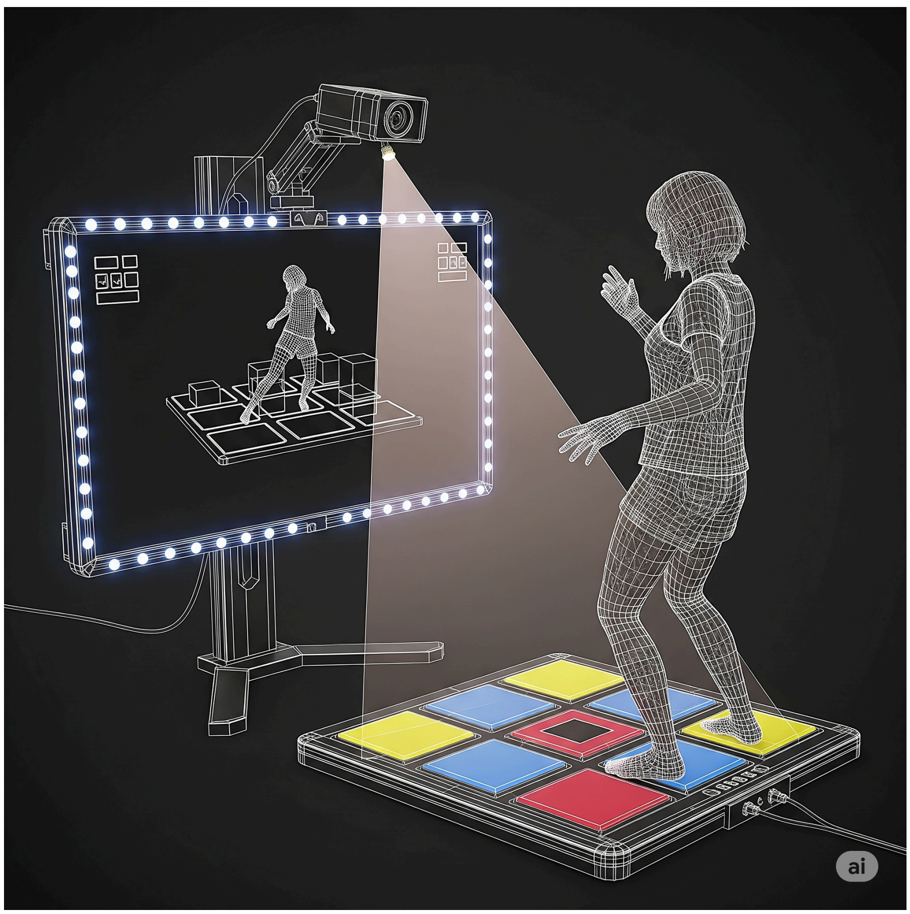


Figure 2.8 *Conceptual Rendering of the S.T.E.P.S Gameplay Environment. A visual overview of the player-camera interaction and LED frame layout. The LEDs are shown as white for illustration purposes, but the final system uses 850nm NIR LEDs. This illustration is not to scale and is intended for conceptual understanding.*

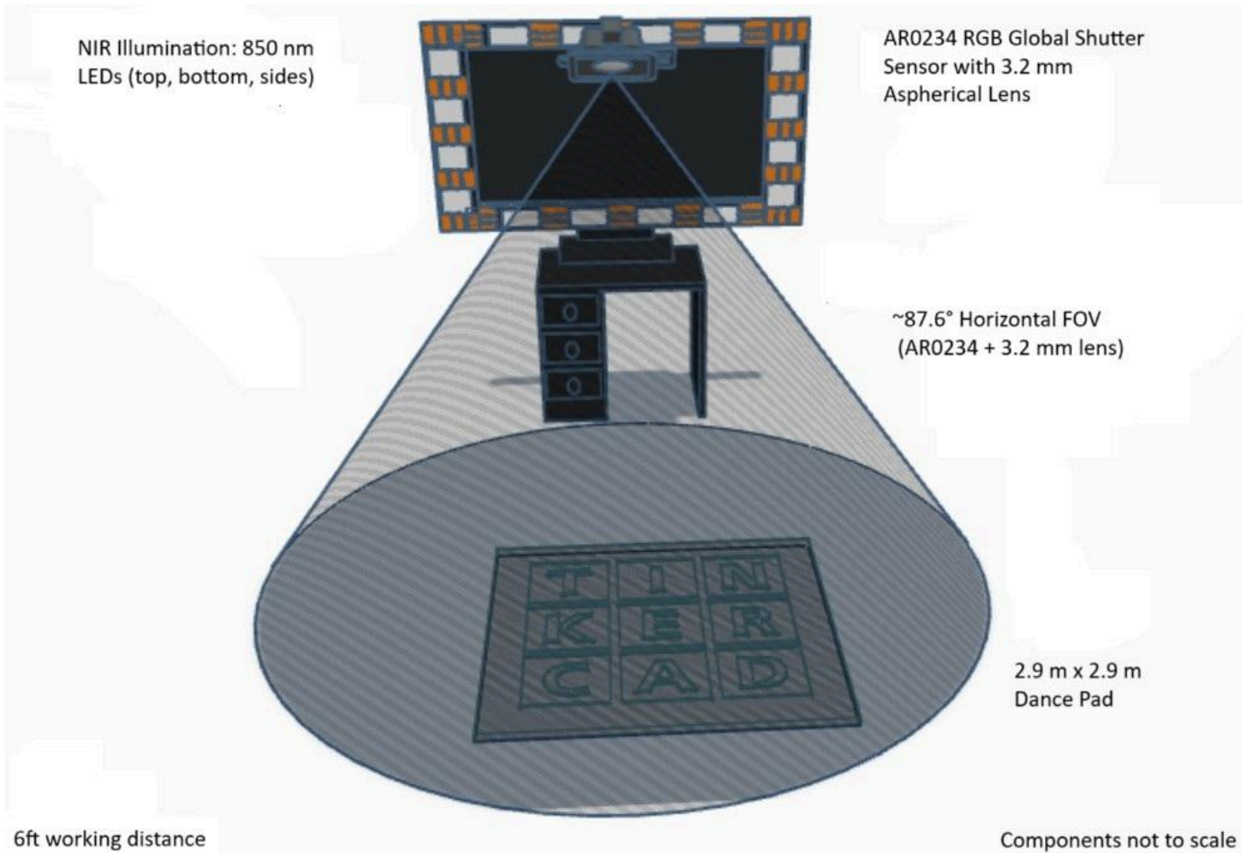


Figure 2.9 *Schematic layout of the optical and illumination subsystems. The AR0234 camera with a 3.2 mm lens captures a ~87.6° FOV across the 2.9 m × 2.9 m tracking zone from 6 ft away. Surrounding 850 nm NIR LEDs illuminate the scene to support reliable pose detection.*

2.5 Project Features and Functionalities

The primary goal of this project is to develop a working proof-of-concept for a new kind of dance rhythm game that integrates physical input with computer vision-based expression tracking. The focus is on delivering a functional and demonstrable system that includes three core components: a responsive 9-panel dance pad, a custom rhythm video game engine, and a vision system that scores player movement based on style and expressiveness.

At the hardware level, the dance pad will use Force-Sensing Resistors (FSRs) beneath each panel to detect foot pressure and translate those inputs into digital signals via a microcontroller. The signals are interpreted as button presses in-game, allowing for responsive gameplay. Each panel will also include LED lighting to provide immediate visual feedback based on the game's state and player interaction. While commercial-grade materials like metal panels or acrylic overlays would be ideal, the immediate priority is to build a stable, functional pad using accessible prototyping materials to prove the input system works reliably.

The software side features a rhythm game engine tailored to the 9-direction input system. Players will step on directional pads in sync with music, guided by scrolling notes on screen. The game will evaluate the player’s timing accuracy and display scores at the end of each round. Alongside traditional gameplay scoring, a connected camera system will assess the player’s full-body movement using a pose detection library. Based on the amplitude, variation, and expressiveness of the player’s dance, the game will generate a secondary “style score.” This adds a creative and engaging layer of performance evaluation beyond pure timing.

A basic user interface will allow for song selection, score displays, and navigation through the system. A chart editor tool will also be included, enabling users to import music and design their own charts either manually or with the help of automatic generation tools.

Although not essential to the proof-of-concept, we also envision the long-term possibility of turning the system into a self-contained arcade-style unit. Features like a cabinet enclosure, co-op integration, player logins, and online leaderboards are considered stretch goals that could be implemented later with more resources. The project is designed with scalability in mind: the core technology should work independently, while leaving room for future upgrades to polish and deploy the system as a full commercial or open-source product.

2.5.1 Style Score and Pose-Based Evaluation System

In addition to the traditional timing-based score system, our game introduces a novel secondary metric known as the Style Score, which rewards players for striking expressive and clearly defined poses at designated times during gameplay. Unlike freeform dance scoring, our system focuses on detecting static full-body poses that can be consistently recognized by a pose estimation algorithm.

We intentionally limit the scope of detection to distinct, predefined poses that can be reliably tracked in real-time using a single camera. This approach improves detection accuracy and reduces computational load while still encouraging expressive movement.

During gameplay, specific pose prompts will appear, similar to freestyle moments or bonus sections. These pose prompts will be displayed in a small icon area on screen during designated freestyle sections, giving players 2–3 seconds to match the target pose. If the player strikes the correct pose at the right time, they are awarded bonus points to their Style Score. This incentivizes physical creativity and rewards players who engage more fully with the visual performance aspect of rhythm games.

Each pose is chosen based on ease of detection, body separation, and iconic visual silhouette (all poses are mirrored when displayed to the player):

- **Muscle Man Pose**: Both arms raised as if you're flexing biceps'
- **What? Pose:** Both arms out as if you don't know about something
- **Point Up Pose:** Left arm behind the head, pointing out to a plane
- **Tough Guy Pose**: Crossed arms
- **Samurai Pose** : Legs wide apart, one hand near waist as if gripping a katana, other arm pointing forward or out.
- **Mantis Pose**: Right arm raised in front of the chest, left bent above the head, and right leg up with knee up

Each pose will have keypoint thresholds that define acceptable angles and positions such as a certain limb being within a range of 20 degrees from the desired state. This will be validated using a pre-trained MediaPipe/OpenPose[5] model.

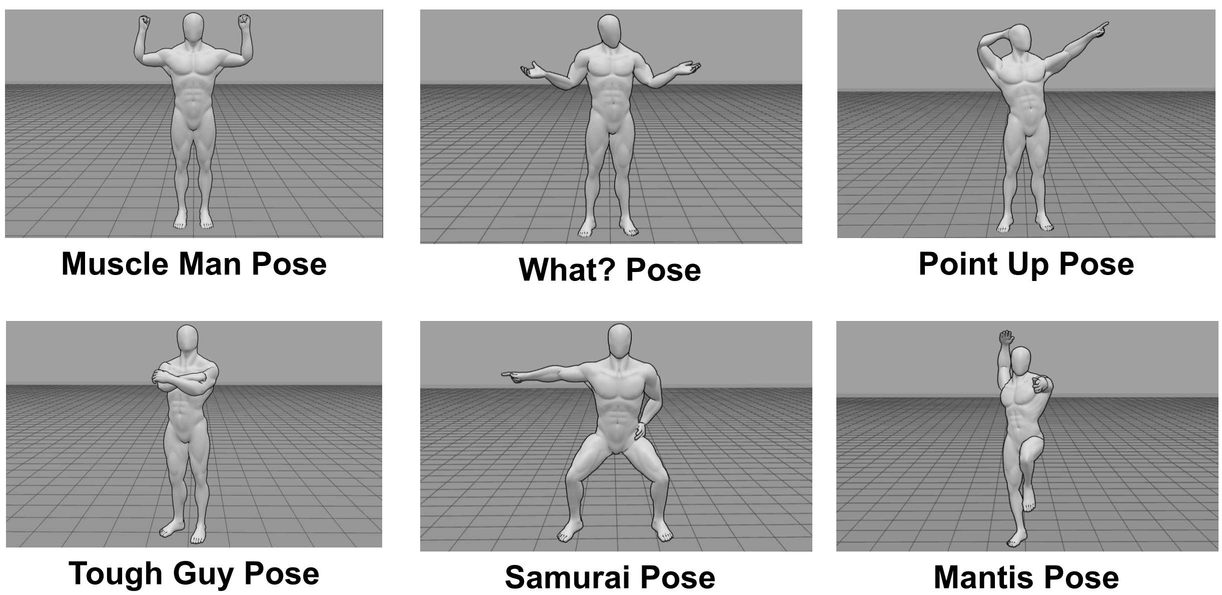


Figure 2.10 *Images of the tentative poses (modeled using PoseMy.Art)*

2.6 Requirements and Specifications

2.6.1 Parts Specification Requirements Table

Table 2.1 *List of all of the project's engineering requirements and specifications. Highlighted in yellow are specifications that will be demonstrated*

Parameter	Value
Overall System	
Active power consumption	~45 W
Lighting response time	~5-10 ms
Pose identification accuracy	≥ 95% detection accuracy for body/limb motion
Input response time	~5-10 ms
Dance Pad	
Size	~34.5 x 34.5 x 2.5 in
Weight	≤ 30 lbs
Cost	≤ $300
Printed Circuit Boards (PCB)	
Size	≤ 10 cm^2
Display	
Frame rate	≥ 60 fps
Resolution	≥ 1280 x 720 pixels
Refresh rate	≥ 120 Hz
Dance Pad Panels	
Size	~10 in^2
FSRs input response time	~5-10 ms
FSRs force range input	0-100 N
RGB LEDs PWM duty cycle	~10%-30%

Camera Module	
Full-body coverage area	≥ 2.9 m field width at 1.83 m distance, ensures full-body coverage with no tracking cutoff
Power Supply Unit	
Input voltage from wall power via AC-DC converter	≥ 12 V
Output power	≥ 60W (≥ 5A @ 12V)
LED Panel	
illumination uniformity of the player	≥ 90% uniformity across player body, achieved by supplemental 850 nm illumination regardless of ambient light
Player visibility (shadow coverage)	Full Body at 1m

2.6.2 Notes on Specifications

<u>Overall System</u>
For active power consumption, most typical dance arcade machines are around 100-200 watts, so for our system we aim to have set our limit to 200 watts.

<u>Dance Pad</u>
For increased portability and potentially affordability and maintenance, a size of about 34.5 x 34.5 x 2.5 inches and weight of at most 30 lbs would be optimal for the player and owner of the pad.

<u>Printed Circuit Board (PCB)</u>
As mentioned before in goals, we want to minimize the size of the PCB as much as we can while also making it easy enough to check for errors on the design.

<u>Display</u>
Whether it be a PC or game system, the system requirements are the same. In order to run our custom made game, the system must be able to run at least 60 frames per second, have a resolution of at least 1280 * 720 pixels, and have a refresh rate of at least 120 Hz.

<u>Dance Pad Panel</u>
Response time from the FSRs must be almost instantaneous and this is very important because rhythm games are dependent on timing of when commands are hit. There should be little to no latency when pressing the pads and triggering the FSRs.

Moreover, the amount of force applied needed to trigger the FSRs must be 0-100 N so that even the smallest amount of pressure should be enough to send a signal back to the MCU.

Camera Module
To meet ≥95% tracking accuracy and full-body coverage, the camera must maintain sufficient resolution, wide FOV (≥ 400 mm at 1m), and minimal distortion. This ensures consistent player visibility without cutoff or blind spots.

Power Supply Unit
The pad is powered through a wall power cable which should be more than sufficient enough to power the pad. The input voltage of at least 12 V is required to efficiently power the whole pad, and we aim to output a power of at least 1.65 watts.

LED Panels
To ensure the player is always fully illuminated regardless of body position or movement, the LED panels must maintain illumination uniformity of ≥ 90% and deliver sufficient brightness to achieve full body visibility at a 1-meter distance.

2.7 Hardware Block Diagram

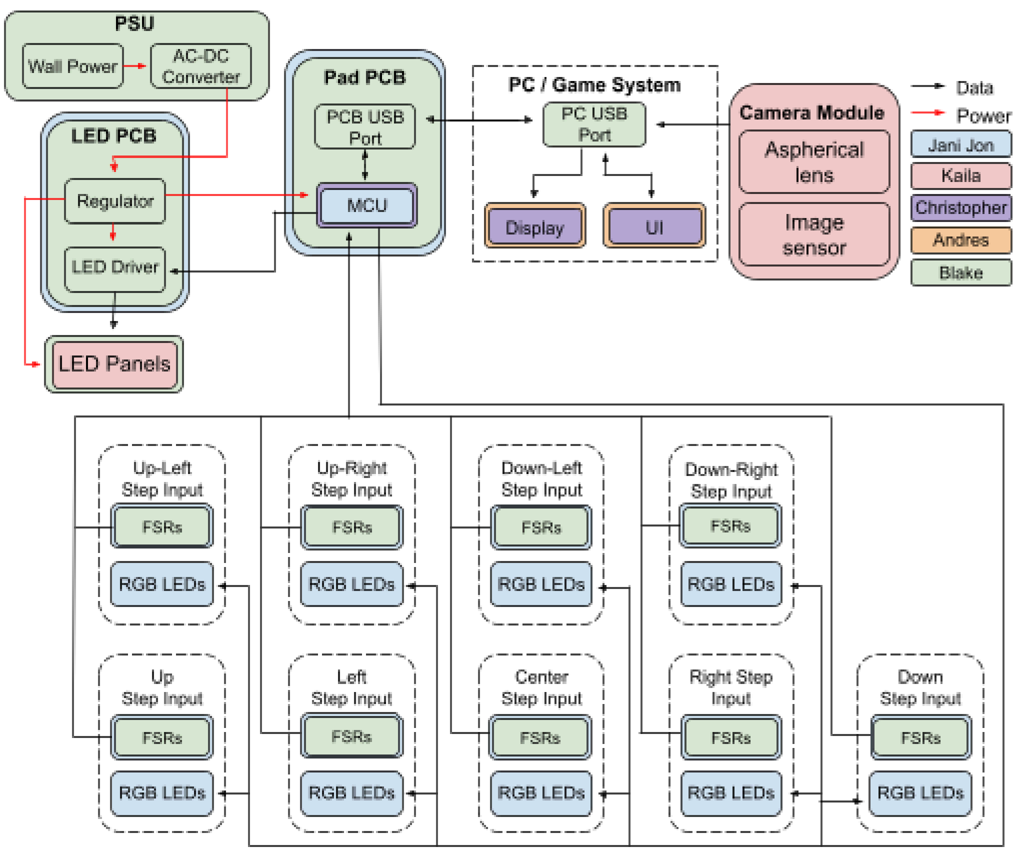


Figure 2.11 *Hardware Block Diagram showcasing work distribution and major components of the design*

2.8 Software Block Diagram

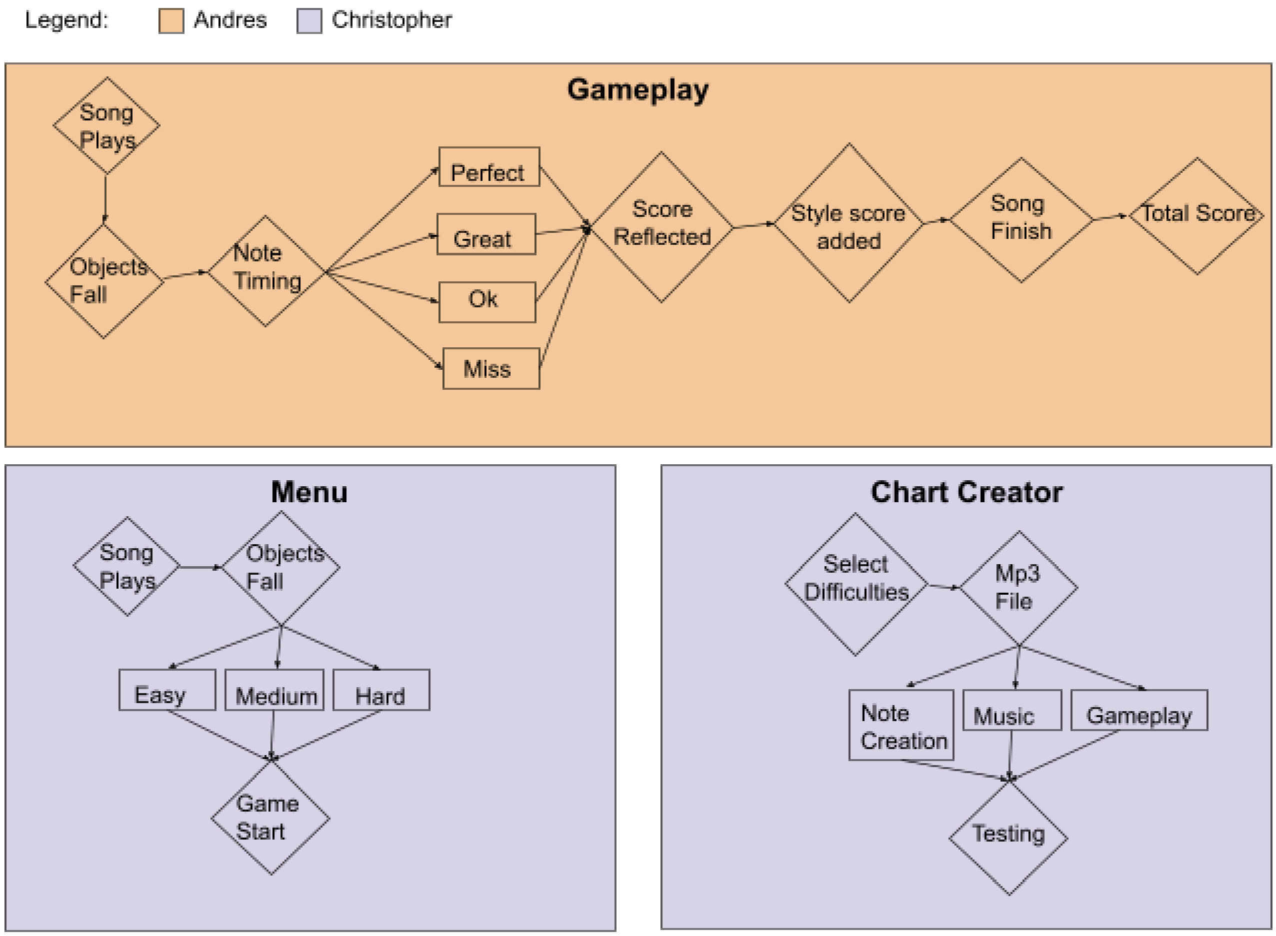


Figure 2.12 *Software Block Diagram of Gameplay, Menu, and Chart Creator*

2.9 House of Quality

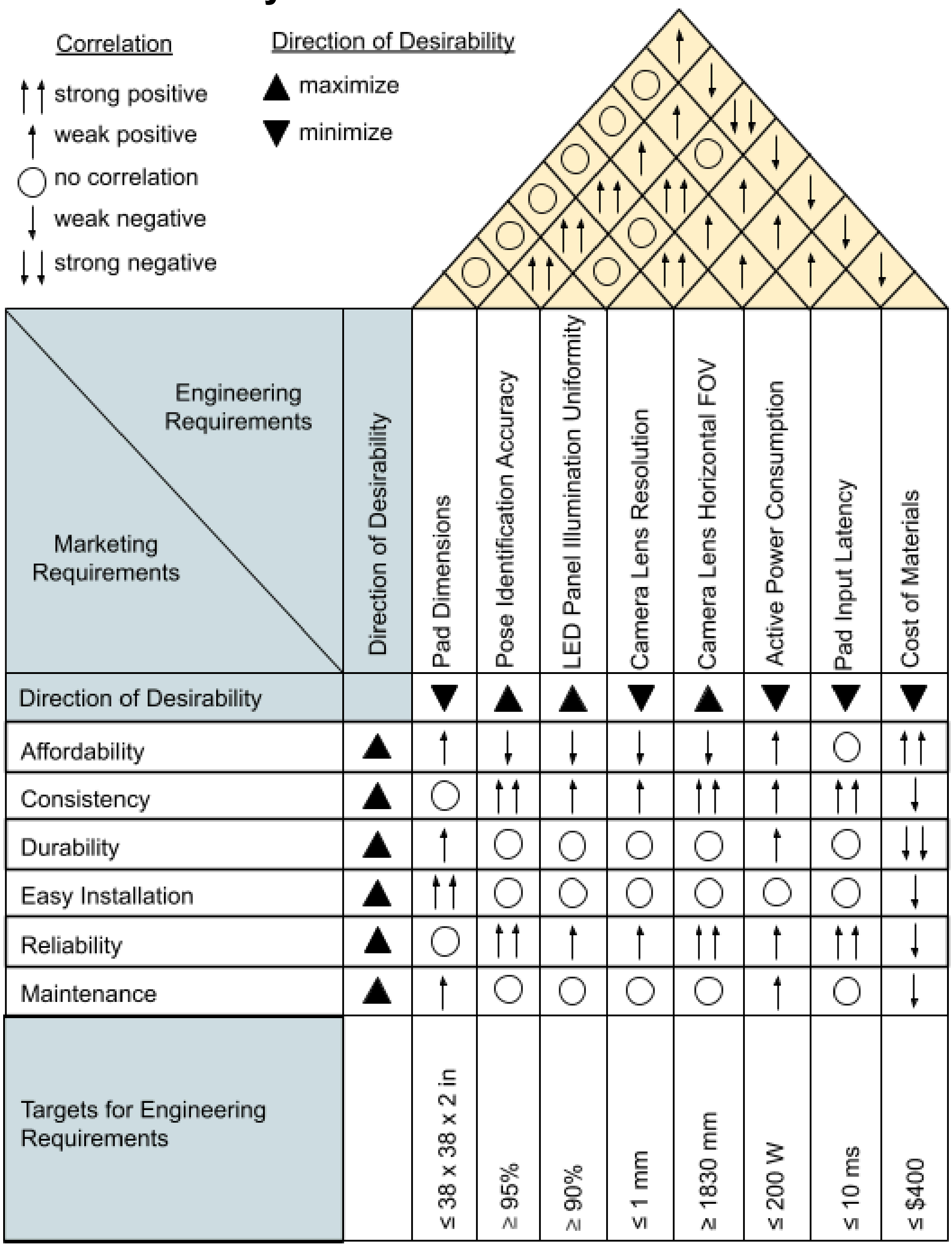


Figure 2.13 *House of Quality*

3. Research

3.1 Embedded Systems

To integrate embedded systems into our STEPS platform, we require a dedicated computing component. The two most suitable options are a microcontroller unit (MCU) or a single-board computer (SBC). Each has its strengths and tradeoffs, but only one is better suited to execute the functionality required for our project.

SBCs offer significantly higher processing power than MCUs, making them capable of running full operating systems, graphical interfaces, and multitasking. However, the tradeoff is that they also consume more power, require more complex infrastructure, and lack real-time response by default. Meanwhile, MCUs are optimized for fast, real-time control, which is critical for rhythm game input responsiveness. They are also cheaper, easier to wire, and consume less power, making them ideal for precise button and LED control. Therefore, we take a deeper dive into both options to evaluate which is most optimal for our dance pad design.

3.1.1 MCU vs SBC

MCUs are designed for task-specific control, particularly those that require low latency and real-time response, such as reading sensor data or controlling actuators. In our system, for input detection, an MCU would allow for minimal system latency, which is a critical factor for timing accuracy in rhythm games.

MCUs also offer faster boot-up times and simpler circuit integration, making them easier to debug and more reliable during development. Most MCUs include a large number of GPIO pins, many with PWM support, which makes them especially suitable for handling LEDs and button inputs with minimal delay. Their hardware simplicity aids in easier troubleshooting and faster hardware prototyping. However, MCUs typically require low-level programming (e.g., in C or C++) and are not capable of natively handling advanced UI, graphics, or audio features. As a result, we would require external modules if we were to implement these into our design.

In terms of cost, most MCU chips cost under $10, making them ideal for scalable, low-cost embedded designs. If the board is damaged or needs revision, the low cost of replacement makes development more forgiving.

In contrast, SBCs offer significantly higher processing power and are well-suited for applications involving graphical interfaces, audio, or wireless communication. They run full operating systems, typically Linux-based, and can be programmed using high-level languages like Python or Java, which simplifies software development.

However, their non-deterministic response time, due to OS-level multitasking, makes them less reliable for real-time input detection. This could introduce latency or jitter in detecting panel hits, affecting the gameplay experience. SBCs also have longer boot times, higher power consumption, and are generally more complex to integrate into custom PCB designs. On top of that, SBCs typically cost between $35–$60, which increases development risk if the board is damaged or requires replacement.

3.1.2 MCU is the Better Option… But Why?

While neither MCUs nor SBCs are perfect in every category, MCUs are the more optimal choice for our dance pad design due to their fast and predictable response times, lower power consumption, smaller footprint, and significantly lower cost. As shown in *Table 3.1* below, the MCU excels in the areas most critical to our engineering and marketing goals, such as system responsiveness, efficiency, and affordability, as previously outlined in *Figure 2.9 House of Quality*.

To meet cost targets, we must use components that are low-cost yet reliable, and MCUs typically cost under $10, making them ideal for both prototyping and scalable production. Power consumption is another key concern, especially since our system will run multiple FSRs and RGB LEDs simultaneously, and MCUs are optimized for energy efficiency, unlike SBCs which consume more power even when idle. Real-time performance is especially critical for a rhythm-based game, where millisecond-level delays can affect gameplay. MCUs provide immediate input handling and low-latency output control, which are difficult to achieve on SBCs running multitasking operating systems without extensive tuning.

While SBCs may offer advantages like better multimedia processing or easier development with high-level languages, they introduce longer boot times, higher costs, and non-deterministic behavior, making them less suitable for a system where precision and timing matter most. MCUs, on the other hand, are straightforward to integrate, faster to initialize, and allow for fine-grained control over every component in the system.

Overall, MCUs give us the performance and reliability needed for fast input detection, smooth LED animations, and efficient control, all while keeping costs, power, and complexity well within our project’s constraints.

Table 3.1 *Comparison table between MCUs and SBCs using their features*

Feature	Microcontroller (MCU)	Single Board Computer (SBC)
Processing Power		✓
Real-Time Response	✓	
Design Complexity	✓	

Coding Complexity		✓
Power Consumption	✓	
I/O Handling	✓	
UI Support		✓
Boot Time	✓	
Cost	✓	

3.1.3 Arduino

Arduino is a series of open-source microcontroller boards designed for ease of use in both hardware and software. They are especially popular in the prototyping and hobbyist community due to their beginner-friendly development environment (Arduino IDE), and extensive library support for I/O, sensors, LEDs, and communication protocols. Arduino boards typically use AVR-based microcontrollers, although some variants use ARM cores. In this section, we look at three of the most fitting boards for our dance pad design: Arduino Uno, Arduino Mega 2560, and Arduino Leonardo.

One of Arduino's most iconic boards is the Arduino Uno, which uses the 8-bit ATmega328P microcontroller. It runs at a clock speed of 16 MHz, with 32 KB of flash memory, 2 KB of SRAM, and 1 KB of EEPROM. The Uno includes 14 digital I/O pins (6 with PWM support) and 6 analog inputs (ADC). It's widely supported and extremely simple to program and wire, making it a go-to for prototyping. However, it lacks native USB HID support, which means it cannot act as a USB joystick or keyboard without external USB-serial bridging. For our dance pad, the Uno would fall short on both ADC and PWM channels unless additional multiplexing components were added.
One of Arduino's higher-end boards is the Arduino Mega 2560, which features the ATmega2560 microcontroller. Like the Uno, it runs at 16 MHz, but includes significantly more memory: 256 KB of flash, 8 KB of SRAM, and 4 KB of EEPROM. It provides 54 GPIO pins, 15 PWM channels, 16 ADC channels, and 4 UART serial ports, making it highly capable of handling our dance pad's inputs. The Mega does not support native USB HID, but its I/O abundance makes it ideal for wiring-heavy applications.

Another one of Arduino's boards is the Arduino Leonardo, which uses the ATmega32U4 microcontroller. It also runs at 16 MHz, with 32 KB of flash memory, 2.5 KB of SRAM, and 1 KB of EEPROM. It provides 20 GPIO pins, 7 PWM channels, and 12 ADC channels. Its standout feature is native USB HID support, allowing it to behave like a USB keyboard or joystick, a feature perfect for our design. While it offers fewer I/O pins than the Mega, it still covers our minimum requirements, and any shortfall can be handled using external components.

Table 3.2 *Arduino boards and their features comparison table*

Feature	Arduino Uno	Arduino Mega 2560	Arduino Leonardo
Main MCU	**ATmega328P**	**ATmega2560**	**ATmega32U4**
Clock Speed	16 MHz	16 MHz	16 MHz
Flash Memory	32 KB	256 KB	32 KB
SRAM	2 KB	8 KB	2.5 KB
EEPROM	1 KB	4 KB	1 KB
GPIO Pins	14	54	20
PWM Channels	6	15	7
ADC Inputs	6	16	12
USB Communication	serial-to-USB	serial-to-USB	native USB
USB HID Support	none	none	included
Serial Ports (UART)	1	4	1
Active Power Consumption	Moderate	Moderate-High	Moderate
MCU Chip Cost	~$3	~$8	~$4

3.1.4 Teensy

Teensy is a powerful and compact microcontroller board series developed by PJRC. While slightly less common than Arduino in beginner projects, Teensy is widely used in performance-critical embedded systems due to its speed, USB capabilities, and real-time performance. Most Teensy boards use ARM Cortex-M cores, which were specifically designed for embedded applications to offer high energy efficiency and low cost. All Teensy boards also support native USB HID, making them excellent for applications that require low-latency input handling, such as rhythm games. In this section, we explore four Teensy models: Teensy 2.0, Teensy 3.2, Teensy LC (Low Cost), and Teensy 4.1.

The oldest version of them all is the Teensy 2.0 (released in 2009). Out of our other Teensy options, Teensy 2.0 is the only one that doesn't use an ARM core. Instead, much like the Arduino Leonardo, it uses an ATmega32U4 8-bit automatic voltage regulator (AVR). This means that the Teensy 2.0 has almost the same exact specifications as an Arduino Leonardo. For instance, both run at a clock of 16 MHz and include 32 KB of flash memory, 2.5 KB of SRAM, 12 ADC channels, and USB HID support. It provides 25 GPIO pins, which is slightly more than the Leonardo's 20. Despite its age, Teensy 2.0 remains a lightweight and reliable choice for simple USB input devices. However, it lacks the processing power and memory needed for advanced LED animations or real-time multitasking.

Released in 2014, the Teensy 3.2 offers a major leap in performance over the 2.0. It features an MK20DX256VLH7 MCU with a 72 MHz ARM Cortex-M4 processor, supporting both DSP instructions and hardware floating point. It includes 256 KB of flash memory, 64 KB of SRAM, and 34 GPIO pins. The Teensy 3.2 supports 21 analog inputs (ADC) and 12 PWM outputs, and unlike Teensy 2.0, the ADC is 13-bit (not 16-bit as sometimes misreported). It supports USB HID, has robust NeoPixel support (including DMA-based LED control), and is powerful enough to handle real-time step detection, LED animations, and game communication simultaneously. It offers an excellent balance between performance, GPIO availability, and price, making it a strong candidate for mid-level embedded designs like our design.

Introduced in 2015, the Teensy LC is a budget-friendly alternative that still offers modern features. It uses the MKL26Z64VFT4 MCU, based on a 48 MHz ARM Cortex-M0+ processor. It comes with 62 KB of flash memory, 8 KB of SRAM, and 27 GPIO pins. It includes 13 analog inputs and 10 PWM outputs, and it supports USB HID. While not as fast as the 3.2, the Teensy LC still outperforms Teensy 2.0 in nearly all areas. It does not support DMA for NeoPixel LED control, meaning LED animations may add CPU overhead. Still, its low cost and decent GPIO make it a practical option if budget is a major constraint.

The most powerful board in the series, Teensy 4.1, was released in 2020 and uses the MIMXRT1062 MCU with an ARM Cortex-M7 processor running at a blazing 600 MHz. It includes 1 MB of SRAM, 8 MB of QSPI flash memory, and supports external PSRAM or flash via expansion pads. It features 55 GPIO pins, 18 analog inputs, and up to 35 PWM outputs, with USB High-Speed (480 Mbps) support and full native USB HID. Unlike Teensy 3.2 or LC, Teensy 4.1 also supports cache, branch prediction, and complex multitasking, making it ideal for fast input response, dynamic LED control, and interfacing with multiple peripherals simultaneously. While the ADC is only 12-bit and the analog pin count is slightly lower than Teensy 3.2, the vast processing power and memory make Teensy 4.1 a top-tier choice.

Table 3.3 *Teensy boards and their features comparison table*

Feature	Teensy 2.0	Teensy 3.2	Teensy LC	Teensy 4.1
MCU Chip	**ATmega32U4**	**MK20DX256VLH7**	**MKL26Z64VFT4**	**MIMXRT1062**
Structure	8-bit AVR	ARM Cortex-M4	ARM Cortex-M0+	ARM Cortex-M7
Clock Speed	16 MHz	72 MHz	48 MHz	600 MHz
Native USB Speed	12 Mbps (full speed)	12 Mbps (full speed)	12 Mbps (full speed)	480 Mbps (high speed)
Flash Memory	32 KB	256 KB	62 KB	8 MB
SRAM	2.5 KB	64 KB	8 KB	1024 KB
GPIO Pins	25	34	27	55
PWM Channels	7	12	10	35
ADC Inputs	12 (12-bit)	21 (13-bit)	13 (12-bit)	18 (12-bit)
Active Power Consumption	Low	Moderate	Low	High-Very High
MCU Chip Cost	~$4	~$6-9	~$2-4	~$7-11

3.1.5 Raspberry Pi Pico (RP2040)

Although there are multiple other Raspberry Pi series like Raspberry Pi 4 and Raspberry Pi Zero, they are not efficient enough to be considered for our design. Due to the fact that most other Raspberry Pi use Linux which can make them heavier on power and boot time, less reliable when it comes to real-time responses, harder to interface with timing-sensitive components, and more complex to integrate into our dance pad's PCB. The series we want to look at, however, is the Raspberry Pi Pico series.

The Raspberry Pi Pico is a microcontroller, not a general-purpose computer, that uses its own microcontroller chip, the RP2040. The RP2040 is a dual-core Cortex M0+ microcontroller chip that's known for being affordable while having a high performance. Because the RP2040 is made of bare silicon, it comes with a QFN-56 package, which supports compact PCB design and high-speed operation, allowing it to have good thermal dissipation, short lead lengths for better signal integrity, and maintain a compact size for better use of space. However, having the QFN-56 package also makes soldering into the PCB board challenging if done by someone with little to no experience.

The Raspberry Pi Pico runs at a clock speed of 133 MHz, making it well-suited for performance-intensive tasks. Much like the Teensy boards, Pico also supports USB HID, though it requires software implementation via libraries like the Pico SDK or CircuitPython, and no external components or converters are needed for HID communication. However, while the Pico provides 26 GPIO pins and 16 PWM channels, it includes only 3 12-bit ADC channels, which is insufficient to directly read analog signals from all 9 FSR-based pads. To support more analog inputs, external analog multiplexers or ADC chips would be needed. In terms of memory, the Pico offers 2 MB of flash and 264 KB of SRAM, which is significantly more than typical Arduino boards and even some Teensy models. Overall, Raspberry Pi Pico's RP2040 is a highly affordable chip (~$1) capable of handling complex tasks at high speeds, while maintaining low to moderate active power consumption. However, using this chip will require external components, such as analog multiplexers for more ADC inputs, for our design to be fully functional.

3.1.6 ESP32

The ESP32 is a series of low-cost, low-power system-on-a-chips (SOCs). MCU chips used by ESP32 modules typically cost around $1-$3. What makes the ESP32 different from traditional microcontrollers like AVR or RP2040 is that most ESP32 modules have Wi-Fi and/or Bluetooth capabilities. In this section, we analyze and compare the ESP32 (WROOM-32), ESP32-S2, ESP32-S3, and ESP32-C3.

The ESP32-WROOM-32 is one of the earliest and most commonly used modules in the ESP32 series. It features a dual-core 32-bit Xtensa LX6 processor and supports both Wi-Fi and Bluetooth Classic + BLE. Despite being one of the older models, it can still run at up to 240 MHz, making it capable of handling performance-heavy tasks. It comes with 520 KB of SRAM and 4 MB of flash memory by default, though variants with 8 MB or 16 MB of flash are also available. The chip provides up to 34 GPIO pins, 16 PWM channels, and 18 ADC channels, which is sufficient for our design. However, one major limitation is that it does not have native USB HID support without additional hardware. To use the WROOM-32 in a USB-connected setup, we would need to add an external USB-to-HID bridge chip or pair it with another microcontroller that supports USB HID natively. Therefore, unless we design the DDR pad to be fully wireless, it may be more practical to use another ESP32 variant that can support native USB HID directly.

The ESP32-S2 improves on the WROOM-32, most notably, by adding native USB support. It features a single-core Xtensa LX7 processor that can run up to 240 MHz, and, by default, includes 320 KB of SRAM and 4 MB of flash memory. Like other ESP32 modules, it supports Wi-Fi, but unlike the original WROOM-32, it does not support Bluetooth. The chip provides up to 43 GPIO pins, 16 PWM channels, and 20 ADC channels, which is slightly more compared to WROOM-32. What sets the ESP32-S2 apart is its native USB OTG support, which allows it to act as a USB HID device, such as a gamepad or keyboard, without requiring any external USB bridge or extra MCU. The main drawback is its single-core design, which limits multitasking performance compared to dual-core models, but this tradeoff may be acceptable if real-time responsiveness is properly managed.

The ESP32-S3 builds upon the capabilities of the S2 and is arguably the most feature-rich variant in the ESP32 family for the design we're aiming to achieve. It retains native USB HID support, much like S2. It uses a dual-core Xtensa LX7 processor running at up to 240 MHz, with 512 KB of SRAM and 128 KB of RTC memory, offering more headroom for handling concurrent tasks such as sensor polling, LED animations, and USB communication. The S3 has 45 GPIO pins, 8 PWM channels, 20 ADC channels, and it expands functionality with BLE 5.0 support (though it lacks classic Bluetooth). The ESP32-S3 is especially suitable for performance-demanding applications that require both USB connectivity and multiple analog inputs, making it one of the best choices for a wired dance pad built around a single microcontroller.

The ESP32-C3 is a compact, low-cost MCU in the ESP32 family that trades raw performance for simplicity and power efficiency. It features a single-core 32-bit RISC-V processor running at up to 160 MHz, with 400 KB of SRAM and 4 MB of flash memory. While some variants of C3 can support external flash chips up to 16 MB, some don't, like the ESP32-C3-WROOM-02U. One of its most attractive features is its inclusion of native USB 2.0 support, enabling it to function as a USB HID device without requiring external USB-to-serial hardware. However, C3 only supports 22 GPIO pins, 6 hardware LED PWM channels, and 6 ADC channels, which makes it unsuitable for reading all of our dance pad's inputs directly without the use of an analog multiplexer or external ADC chip. It does support BLE 5.0, though it lacks classic Bluetooth. Overall, while it's not as powerful or feature-rich as the S2 or S3, the ESP32-C3 is a budget-friendly option for our designs, especially if analog input limitations can be mitigated with external circuitry.

Table 3.4 *ESP32 boards and their features comparison table*

Feature	ESP32-WROOM-32	ESP32-S2	ESP32-S3	ESP32-C3
MCU Chip	**ESP32-D0WD**	**ESP32-S2**	**ESP32-S3**	**ESP32-C3**
Structure	Dual-core Xtensa LX6	Single-core LX7	Dual-core LX7	Single-core RISC-V
Clock Speed	240 MHz	240 MHz	240 MHz	160 MHz
Flash Memory	4-16 MB	4-16 MB	4-16 MB	4-16 MB (some)
SRAM	520 KB	320 KB (128 KB RTC)	512 KB (128 KB RTC)	400 KB
GPIO Pins	34	43	45	22
PWM Channels	16	16	8	6
ADC Inputs	18	20	20	6
Native USB	None	USB OTG	USB OTG	USB 2.0
Wi-Fi	Yes	Yes	Yes	Yes
Bluetooth	Classic + BLE	None	BLE 5.0	BLE 5.0
Active Power Consumption	High	Moderate-High	High	Moderate
MCU Chip Cost	~$2-3	~$2	~$2.50-3	~$1.50-2

3.1.7 STM32

STM32 is a series of 32-bit ARM Cortex-M microcontrollers. They're known for having long-term reliability, rich peripheral sets (i.e. ADC, USB, timers, etc.), native USB support in most of its variants, strong real-time performance, great power efficiency, and full support in IDEs like STM32CubeIDE, Keil, PlatformIO, and Arduino (for some variants). We will be looking at the most common and fitting variant from each of the following STM32 families: STM32F1, STM32F4, STM32F0, and STM32L4.

One of STM32F1's most common MCU chips is the STM32F103C8, also known as the "Blue Pill" chip. It features a 72 MHz Cortex-M3 processor, 64 KB of flash memory, and 20 KB of SRAM. It also has 37 GPIO pins, 15 PWM channels, and 10 ADC channels, which is sufficient for our design requirements. STM32F103C8 supports native USB Full-Speed, which can be configured for HID communication using STM32CubeMX and the HAL libraries. However, being an older generation chip, it has limited memory for larger LED effects or buffered inputs. Still, it remains a cost-effective and capable option for our design.

One of STM32F4's most common MCU chips is the STM32F407VG, which is a high-performance microcontroller based on the 168 MHz Cortex-M4 core with DSP and FPU support. It includes 1 MB of flash memory and 192 KB of SRAM. It also has 82 GPIO pins, 3 ADCs (12-bit) that can handle up to a total of 16 multiplexed input channels, and supports up to 17 timers, several of which can generate PWM signals with up to 4 channels per timer. It also supports both Full-Speed and High-Speed USB, with High-Speed USB requiring an external PHY (Physical Layer). Overall, the STM32F407VG is ideal for advanced dance pad designs aiming for high responsiveness, complex LED patterns, or additional input features. The tradeoff is higher cost, larger size, and slightly more power consumption.

One of STM32F0's most common MCU chips is the STM32F072RB, which features a 48 MHz Cortex-M0 processor, 128 KB of flash memory, and 16 KB of SRAM. It includes 51 GPIO pins, 18 PWM channels, and 16 ADC channels (12-bit), making it a capable and low-power choice for handling our dance pad inputs and driving RGB LEDs. STM32F072RB supports native USB Full-Speed, and can be configured as a USB HID device using STM32CubeMX and the HAL libraries. While it lacks advanced processing features like DSP or FPU, its simplicity, low-cost, low power consumption, and USB support make it a great option for our design.

One of STM32L4's most versatile MCU chips is the STM32L476RG, which uses an 80 MHz Cortex-M4 core with both DSP and FPU support. It comes with 1 MB of flash memory and 128 KB of SRAM, offering plenty of space for real-time processing and buffering. The STM32L476RG provides up to 76 GPIO pins, 24 PWM channels, and 16 ADC channels (12-bit), allowing for extensive input reading and precise LED control. It also supports native USB Full-Speed and can operate in low-power modes, making it suitable for our design as it helps improve both performance and energy efficiency. The main trade off is its slightly higher cost, but its extensive peripheral set and low-power capabilities make it a strong candidate for our design.

Table 3.5 *STM32 common MCU chips from different variants comparison table*

Feature	STM32F1	STM32F4	STM32F0	STM32L4
MCU Chip	STM32F103C8	STM32F407VG	STM32F407VG	STM32L476RG
Structure	Cortex-M3	Cortex-M4	Cortex-M0	Cortex-M4
Clock Speed	72 MHz	168 MHz	48 MHz	80 MHz
Flash Memory	64 KB	1 MB	128 KB	1 MB
SRAM	20 KB	192 KB	16 KB	128 KB
GPIO Pins	37	82	51	76
PWM Channels	15	~17 timers each ≤ 4 PWM	18	24
ADC Inputs	10	16	16	16
Native USB	FS USB	FS USB + HS (w/ PHY)	FS USB	FS USB
Active Power Consumption	Moderate	High	Very Low	Very Low
MCU Chip Cost	~$2	~$7-10	~$1.50-2	~$4-5

3.1.8 MCU Selection

After comparing a wide range of microcontroller options, the Arduino Leonardo is the most optimal choice for our dance pad system based on its balance of cost-efficiency, simplicity, and native USB HID support. While it doesn't offer the same raw processing power or extensive I/O capacity as higher-end MCUs like Teensy 4.1 or STM32F4 variants, the Leonardo meets all of our core requirements while keeping system complexity low.

Arduino Leonardo is powered by an 8-bit ATmega32U4 processor running at 16 MHz, with 32 KB of flash memory, 2.5 KB of SRAM, 30 I/O pins (7 PWM), and 12 ADC inputs. It natively supports USB HID, which allows it to act as a keyboard or joystick, crucial for our game's real-time step detection. Compared to other Arduino boards like Uno or Mega, Leonardo is better suited to our needs due to this built-in USB functionality, eliminating the need for external USB-to-serial converters.

Although the Leonardo offers fewer GPIO and less memory than boards like Teensy 4.1 or ESP32, its performance is still sufficient for handling 9 FSR sensors and RGB LEDs, especially with efficient software handling and power management. Additionally, its popularity in DIY projects and compatibility with a wide variety of Arduino libraries make it an accessible and well-documented choice for rapid development and testing.

Overall, Arduino Leonardo strikes the right balance between functionality, ease of use, and affordability, making it a practical and reliable core controller for our dance pad system, especially for single-player setups with limited hardware demands.

3.2 Dance Pad Sensor

In order for the pad to send signals to the game, we need sensors that can detect physical input and convert it into electrical signals for the microcontroller to process. There are four common sensor types we can consider for this project: force-sensing resistors (FSRs), load cells, strain gauges, and piezoelectric sensors. We compare them based on cost, accuracy, complexity, size, durability, and their ability to detect both hold steps and taps. First, we'll explore what each sensor is, how it works, and how it meets the project's engineering requirements.

3.2.1 Force Sensing Resistors

Force-sensing resistors (FSRs) are sensors that change resistance in response to applied pressure. They are commonly used in DIY and custom dance pad projects because they are inexpensive, easy to wire, and can be constructed using accessible materials like Velostat or copper tape. FSRs are also thin, flexible, and lightweight, making them ideal for compact pad designs.

Though FSRs offer quick response times and simple analog interfacing (via a voltage divider and an ADC pin), they lack precision and can show non-linear output and signal drift over time. Poor placement or surface inconsistency can lead to dead zones or inconsistent detection. Additionally, they are sensitive to temperature and humidity, affecting their long-term reliability. Overall, FSRs are a low-cost, simple, and responsive solution for detecting taps and light pressure in dance pads, but may not be ideal for applications requiring high durability or accuracy over time.

3.2.2 Load Cells

Load cells are force transducers that measure weight or applied force by detecting mechanical deformation, typically using internal strain gauges. They output either a millivolt signal (analog) or, in some designs, a digital signal after amplification. Load cells are known for their high accuracy, excellent stability, and long-term durability.

Their downside lies in their higher complexity: they require a stable mechanical mounting, precise calibration, and amplification circuitry (such as an HX711 module) to interface with a microcontroller. Load cells also tend to be more expensive and bulky, which may be a constraint for slim pad designs.

In summary, load cells are ideal for high-precision, arcade-grade dance pads where accuracy and reliability are top priorities, but they involve more cost, bulk, and circuit complexity than other options.

3.2.3 Strain Gauges (Raw)

Strain gauges are sensing elements that detect strain (deformation) in a material. This is typically a foil or wire pattern that changes electrical resistance when stretched or compressed. Unlike load cells, raw strain gauges don’t measure force directly, but rather the strain on a structure, which can then be correlated to force.

When bonded carefully to structural parts of the pad, strain gauges can be very precise and customized for specific points of interest. However, they are also extremely sensitive to environmental noise and temperature, and they require precise installation, bridge circuits (e.g. Wheatstone bridge), and amplification to produce usable signals.

While strain gauges can be affordable, they are technically complex, and not ready-to-use out of the box. They are best suited for custom mechanical frames where advanced integration is possible and where precision and internal structural feedback are desired.

3.2.4 Piezoelectric Sensors

Piezoelectric sensors use the piezoelectric effect to convert mechanical stress into electrical charge. When pressure or impact is applied, the crystal inside the sensor generates a voltage spike. Piezo sensors are exceptional for detecting fast impacts, such as dance pad taps, and can distinguish between light and hard hits due to their high sensitivity and fast response.

However, they cannot detect sustained pressure or holds, as they only respond to changes in force, not constant force. They are also prone to false triggers from vibration, footstep echoes, or mechanical noise, and they require careful physical isolation or dampening. On the hardware side, they are easy to wire (typically needing just a series resistor and ADC input), though voltage spikes may require clamping diodes to protect the MCU.

Overall, piezoelectric sensors are affordable, fast, and ideal for games focused solely on tap detection, but they are not suitable for designs requiring hold step input or highly stable measurements.

3.2.5 Break Beam Sensors

Break beam sensors are non-contact optical sensors that detect the presence or absence of an object by interrupting a beam of infrared (IR) light between a transmitter and a receiver. When the beam is unbroken, the sensor outputs a steady signal; when the beam is interrupted (e.g. by a footstep), the signal changes, allowing the microcontroller to detect an input event. Break beam sensors are very fast, have high sensitivity, and are not affected by pressure wear, making them highly durable.

However, they cannot detect how hard or long the player is pressing, so they are not capable of reading analog force or hold steps. Alignment between the emitter and receiver must also be precise, and ambient IR sources (like sunlight or reflective surfaces) can interfere with the sensor's reliability. Additionally, their installation typically requires mounting hardware and unobstructed space under or around the panel.

Overall, break beam sensors are a reliable and low-maintenance solution for detecting quick taps or triggered events, but they lack the analog depth needed for pressure-sensitive gameplay and cannot detect holds, making them best suited for tap-only rhythm games or basic input triggering.

3.2.6 Dance Pad Sensor Selection

Upon analyzing all contending sensors for our project, the most optimal choice is clearly the force-sensing resistor (FSR). FSRs are the most commonly used sensors in both DIY dance pad builds and even in many modern arcade dance machines. They are affordable and reliable components that offer fast response times, are simple to wire (usually requiring just a voltage divider circuit), and can detect both types of inputs required by our system: quick impacts and sustained press-and-hold steps.

Their flexibility and thin profile also make them ideal for fitting under lightweight or low-profile panel designs. As long as our physical construction ensures proper placement and force distribution, we can effectively minimize their main drawbacks: potential dead zones, non-linearity, and slightly lower accuracy compared to load cells or strain gauges. Another major advantage of FSRs is cost. Being relatively inexpensive, they allow for cost-effective replacements if damaged during testing or extended use, keeping long-term maintenance costs low. This aligns with our project's goals of affordability and maintainability.

FSRs can be easily wired and pair well with Arduino Leonardo. Much like Arduino Leonardo, the FSR’s simplicity contributes to lower system complexity, enabling us to focus more on refining input responsiveness, game logic, and RGB LED feedback. Overall, FSRs offer the best balance of performance, simplicity, and cost for our dance pad sensor needs.

***Table 3.6** Dance Pad Sensors and their features comparison table*
(Legend: ↑↑ = very high; ↑ = high; M = moderate; ↓ = *moderate/low;* ↓↓ = *very low)*

Feature	FSRs	Load Cells	Strain Gauges (Raw)	Piezoelectric Sensors	Break Beam Sensors
Size	↓↓	↑	↓↓	↓	M
Cost	↓↓	↑	↓	↓	↓
Response Time	↑	↑	↑	↑↑ (impact)	↑↑
Accuracy	↓	↑↑	↑↑	↑ (impact)	↓
Complexity	↓↓	↑	↑↑	M	↓
Durability	↓	↑↑	M	↓	↑
Impact Input	✓	✓	✓	✓	✓
Sustain Input	✓	✓	✓		

3.2.7 Force Sensing Resistor Selection

Now that we have decided on using FSRs, what's the most optimal way to integrate them into our design? There are two ways we can implement the functionality of FSRs into our dance pad:

1. Using the hard physical component FSR Model 408 (300mm length)
2. DIY with conductive material (e.g., copper or aluminum) + Velostat

The FSR Model 408 is a commercially manufactured force-sensing resistor that is 300mm long and designed to offer reliable force measurements in a slim, flexible form. It comes with a pressure-sensitive strip and integrated terminals, making it easy to wire into an analog input. Because it is pre-calibrated and professionally manufactured, it ensures consistency in pressure sensitivity and response across all pad panels. Its reliable construction reduces the risk of false triggers, drift, and dead zones. However, the downside is its higher cost, with each unit typically ranging from $15 to $30 depending on the supplier. In our case, we are looking into Interlink Electronics: $27.93 subtotal + $9.99 shipping = $37.92 total cost. Using nine of these for a full pad setup can add significant expense to the overall budget. Fortunately, we already have five in hand, but we still need to purchase 13 more. It's also slightly less customizable in terms of shape and contact area compared to DIY designs.

The DIY method involves using copper or aluminum tape as conductive contacts with Velostat or other pressure-sensitive films in between. This approach offers high flexibility and full control over the size, shape, and placement of each sensor. It's significantly more cost-effective, for instance, components like Velostat and copper tape are inexpensive and widely available. However, DIY FSRs require careful assembly and testing, as inconsistency in spacing, adhesion, or material thickness can lead to issues like input dead zones or uneven sensitivity. Additionally, they may degrade faster over time compared to commercial FSRs and require more trial and error to calibrate effectively.

Given our goals of reliability, performance consistency, and minimizing potential issues like dead zones or calibration errors, the FSR Model 408 is the optimal choice for our dance pad system. While the DIY conductive method offers more flexibility and lower upfront cost, it also introduces variability and requires more effort in calibration and maintenance. The FSR Model 408 provides a plug-and-play experience with consistent force detection across all panels, reducing uncertainty during development and improving long-term durability. This makes it a better fit for our design, where input timing accuracy and sensor stability are critical for gameplay. The investment in pre-built FSRs will ultimately pay off in performance and ease of integration.

Table 3.7 *FSR Model 408 and DIY Conductive Material + Velostat comparison table (Legend: ↑ = high;* M = moderate; ↓ = *moderate/low)*

Feature	FSR Model 408 (300 mm)	DIY Conductive Material + Velostat
Cost per Sensor	↑ ($15-$30)	↓ (~$3)
Customizability	↓	↑
Assembly Effort	↓	↑
Reliability	↑	M
Sensitivity Consistency	↑	M
Dead Zone Risk	↓	M
Durability	↑	↓
Ease of Replacement	M	↑

3.3 Communication Protocols

3.3.1 External Communication Protocols

The Pad PCB communicates with the PC using the USB (Universal Serial Bus) interface. This connection allows for two-way communication: the pad sends input signals from the FSRs to the game on the PC, while the game may send output feedback such as score or event triggers back to the pad, which can influence the RGB LED animations on the dance pad. This is the only part of the system where a formal communication protocol needs to be noted, since it involves a standardized method of digital data exchange between two systems. Other parts of the system either use simple analog sensing (e.g., FSRs) or digital HIGH/LOW signaling (e.g., to toggle the LED PCB), which do not involve a full protocol.

The communication over USB is handled using a standardized communication protocol, typically either USB CDC (Communication Device Class) or USB HID (Human Interface Device). CDC presents the pad as a virtual serial (COM) port, allowing for general-purpose data exchange. HID, on the other hand, is commonly used for devices like keyboards, mice, and game controllers, and it allows input data (such as button presses or sensor values) to be transmitted to the PC with low latency and without requiring custom drivers. For our design, we are going to use USB HID because we want our pad to act like a keyboard for the PC.

3.3.2 Internal Communication Protocols

In any distributed embedded system, a reliable communication protocol is the essential link that allows multiple microcontrollers and peripherals to work together as a cohesive whole. A communication protocol establishes a set of rules for data exchange, defining everything from the physical number of wires required to the format of the data packets being sent. For the modular 9-tile dance pad, the choice of protocol is a critical design decision. It must efficiently connect the central "Master Controller" board to the nine smart “Input Tiles”, enabling the master to poll for sensor data and send lighting commands with minimal latency and wiring complexity. This section explores the technology behind three of the most common embedded communication protocols: UART, SPI, and I2C compare their strengths and weaknesses and select the most appropriate protocol for the project's specific requirements.

3.3.2.1 UART (Universal Asynchronous Receiver-Transmitter)

UART is a simple and widely used protocol for point-to-point serial communication. It is "asynchronous" because it does not use a shared clock signal to synchronize the sender and receiver. Instead, both devices must be pre-configured to operate at the same data rate, known as the "baud rate."

Communication requires only two wires, a Transmit (TX) pin on one device connects to the Receive (RX) pin on the other. This allows for full-duplex or two-way communication. Data is transmitted in a "frame," which consists of a start bit to signal the beginning of a transmission, 5 to 9 data bits, an optional parity bit for error checking, and one or two stop bits to signal the end of the frame. The idle state of the line is high voltage, and the start bit is a transition to low, which allows the receiving device to synchronize its timing to the incoming data.

Suitability for Project

While UART is simple and reliable for connecting two devices, it is fundamentally a point-to-point protocol. It is not designed to support multiple devices on the same bus without additional complex hardware and software to manage addressing and bus contention. To connect the Master Controller to nine separate Tile Boards, it would require nine separate UART interfaces on the master, which is not practical. Therefore, UART is unsuitable for this project's multi-device bus architecture.

3.3.2.2 Inter-Integrated Circuit (I2C)

I2C is a synchronous, multi-master, multi-slave communication protocol designed specifically for connecting multiple devices on a single, shared bus. It achieves this with remarkable efficiency, requiring only two wires, SCL (Serial Clock): The clock signal, typically generated by the master, SDA (Serial Data): The bidirectional data line.

Unlike SPI, I2C does not use individual slave select lines. Instead, it uses an addressing scheme. Every device on the I2C bus must have a unique 7-bit address. When the master wants to communicate, it first sends out the unique address of the slave it wants to talk to. All devices on the bus hear this address, but only the one whose address matches will respond. This allows a single master to communicate with up to 112 different slave devices using the same two wires.

Suitability for Project

I2C is an excellent protocol for the dance pad's internal bus architecture. Its primary advantage is the ability to connect all nine "Input Tile" boards to the "Master Controller" using just two wires (SCL and SDA), plus power and ground. This drastically simplifies the wiring harness and reduces the required pin count on the master controller when compared to SPI. While its data speed is lower than SPI's, it is more than sufficient to handle polling sensor data and sending lighting commands across the nine tiles without introducing any noticeable latency. Given that the I2C protocol was specifically designed for this type of multi-peripheral, intra-system communication, it is the best engineering choice for this project.

3.3.2.3 Serial Peripheral Interface (SPI)

SPI is a synchronous, full-duplex, master-slave communication protocol known for its very high speed. It is "synchronous" because the master device generates a clock signal that is shared by all devices on the bus, ensuring perfect data synchronization.

A standard SPI bus requires four wires: SCLK (Serial Clock): The clock signal generated by the master. MOSI (Master Out, Slave In): The line the master uses to send data to slaves. MISO (Master In, Slave Out): The line the slaves use to send data back to the master. SS (Slave Select): The master uses a separate SS line for each slave. To talk to a specific slave, the master pulls that slave's SS line low.

This individual slave select line is a key feature of SPI. While it guarantees there are no address conflicts, it also means that connecting to many slaves requires many extra pins on the master device.

Suitability for Project

SPI's primary advantage is its speed, which is often overkill for an application like a dance pad where data packets are small and infrequent. Its main disadvantage for this project is the high pin count. To connect to nine Tile Boards, the Master Controller would need nine dedicated Slave Select pins in addition to the three shared bus lines (SCLK, MOSI, MISO). While the ATmega32U4 has enough pins, this creates a complex and cumbersome wiring harness. For these reasons, SPI is a viable but not the optimal choice.

3.3.3 Comparison and Final Selection

Table 3.8 *Comparison table of communication protocols*

Feature	UART	SPI	I2C
Pin Count	2 (per device)	4 + (N slaves)	2 (total)
Speed	Low	Very High	Moderate
Multi-Device Support	No (Point-to-Point)	Yes (High pin count)	Yes (Address-based)
Complexity	Low	Moderate	Moderate
Project Suitability	Unsuitable	Possible but not ideal	Optimal

The analysis shows that I2C is the optimal communication protocol for the dance pad project. Its primary advantage is its ability to connect all nine Tile Boards to the Master Controller using only two wires for data and clock, plus two for power. This dramatically simplifies the wiring harness, reduces the pin count on the master, and makes the entire system more modular and scalable. While its data speed is lower than SPI's, it is more than fast enough to handle the polling of sensor data and the sending of lighting commands without introducing any noticeable latency. The I2C protocol was specifically designed for this type of intra-system communication, making it the best engineering choice for linking the “Master Controller” to its nine “Input Tile” peripherals.

3.4 Analysis of 12V Power Supplies for Embedded Systems

3.4.1 The Critical Role of the Power Supply

In any electronics project, the power supply is the foundational component upon which all other systems depend. Its primary function is to convert electrical energy from a source, typically a high-voltage AC wall outlet, into a stable, low-voltage DC form that is usable by sensitive electronic components like microcontrollers, sensors, and LEDs. For the 9-tile dance pad project, the power supply is not merely an accessory; it is a critical subsystem that must reliably power both low-current, noise-sensitive logic and high-current, noise-generating lighting systems. A poorly chosen or inadequately designed power supply can lead to a host of problems, including system instability, inaccurate sensor readings, and even permanent damage to components. This paper will explore the technologies behind common 12V power supplies, compare their suitability for the project, and recommend a specific type to ensure robust and reliable operation.

3.4.2 Project Power Requirements

The dance pad project presents a mixed-load challenge. The system requires a single 12V DC input, which must then be conditioned and distributed to serve two distinct functions. The primary load is the high-current 12V infrared (IR) LED strip that surrounds the television display. This high-power load is essential for the computer vision system and will draw approximately 2.8 Amps. The second load is the low-current controller logic, which includes the custom "Power Hub" PCB and its 5V regulator. This regulator powers the main "Master Controller" board and the nine "Tile Board" microcontrollers, representing a low-power, noise-sensitive system that requires a very clean and stable power source.

The total estimated current draw from the 12V source is approximately 3.3 Amps. Therefore, a power supply with a rating of at least 4-5 Amps is required to provide a safe operating margin.

3.4.3 Power Supply Technologies

There are three primary categories of AC-to-DC power supplies that could be considered for this project: unregulated, linear regulated, and switch-mode.

Unregulated Power Supplies

An unregulated power supply is the simplest form of AC-to-DC conversion. It consists of a transformer to step down the AC voltage, a full-wave bridge rectifier to convert the AC to pulsating DC, and a large filter capacitor to smooth the pulsations into a relatively steady DC voltage. While this design is simple, inexpensive, and reliable due to its low component count, its output voltage is not regulated. It will vary significantly with changes in the AC line voltage and the amount of current being drawn. The output also contains significant "ripple," which is leftover AC pulsation that can interfere with sensitive electronics. Due to the lack of regulation and high ripple, this type of supply is unsuitable for the project, as it would be impossible to reliably power the microcontrollers and could cause the LEDs to flicker or dim under load.

Linear Regulated Power Supplies

A linear supply begins with the same components as an unregulated supply but adds a linear regulator. This regulator acts like a variable resistor, constantly adjusting itself to burn off excess voltage as heat, resulting in a very stable and clean output voltage. The primary advantage of this technology is its extremely low noise and ripple, making it ideal for powering highly sensitive analog circuits, audio equipment, and laboratory instruments. However, it is also extremely inefficient. The process of dissipating excess voltage as heat is incredibly wasteful. For this project's power level, a linear supply would be massive, heavy, and would generate a tremendous amount of heat, making it completely impractical. While the clean output is desirable, the poor efficiency and large size render it unsuitable for this application.

3.4.4 Switch-Mode Power Supply (SMPS) Technology

How They Work

A Switch-Mode Power Supply (SMPS) is a much more modern and complex design that achieves very high efficiency. Instead of burning off excess voltage as heat, it works by rapidly switching the input voltage on and off. The process begins by rectifying the incoming AC voltage to high-voltage DC. A high-frequency switch, typically a MOSFET, then chops this DC into a series of pulses thousands of times per second. These high-frequency pulses are fed into a very small, lightweight transformer to be stepped down. The output of the transformer is then rectified and filtered to produce the final, stable output voltage. A feedback control circuit constantly monitors the output and adjusts the switching to keep the voltage perfectly stable, regardless of changes in load or input voltage.

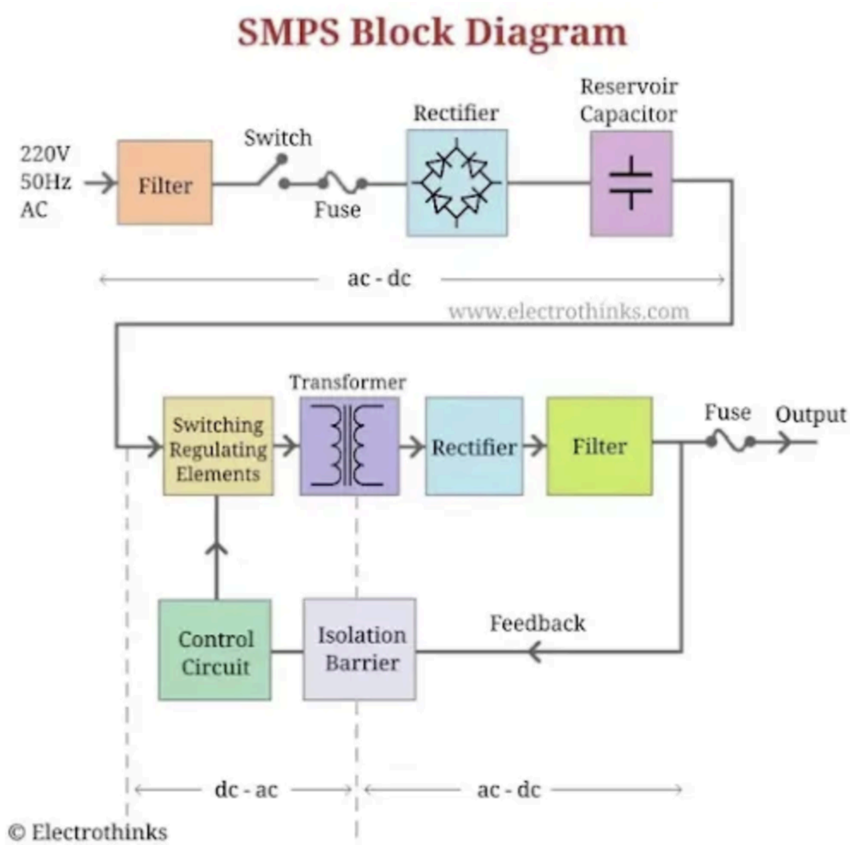


***Figure 3.1** SMPS Block Diagram*

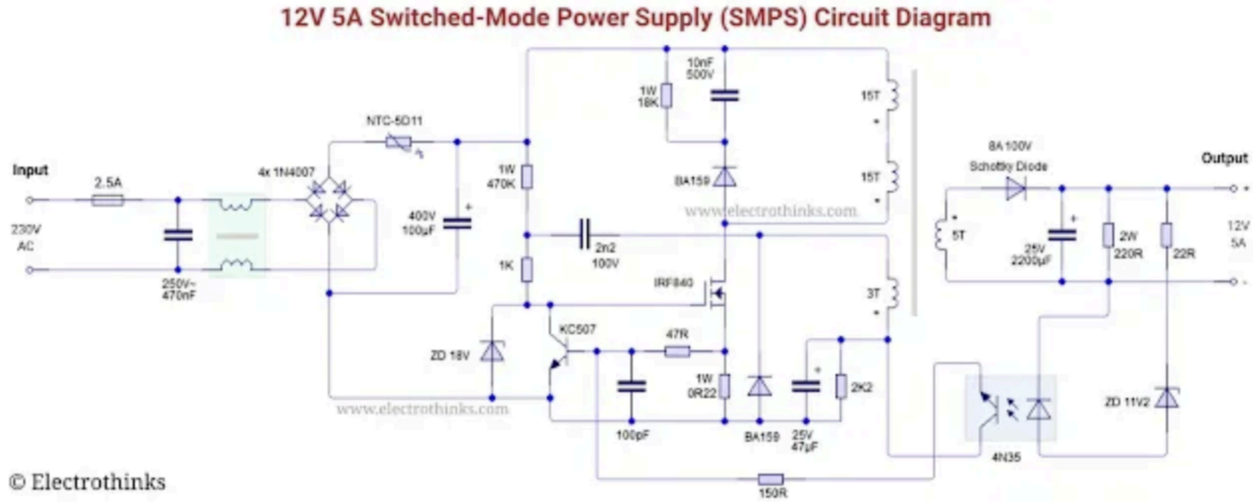


Figure 3.2 *12V 5A SMPS Circuit Diagram*

Characteristics

The primary advantage of an SMPS is its high efficiency, typically between 80-95%, which means very little energy is wasted as heat. The use of a high-frequency transformer allows it to be much more compact and lightweight than a linear supply of the same power rating. Furthermore, most SMPS units can accept a universal AC input, making them usable worldwide. The main drawback is the complexity of the internal design and the electrical noise (EMI/RFI) generated by the high-frequency switching process. However, a well-designed SMPS includes internal filtering to minimize this noise.

Suitability for Project

The SMPS is highly suitable for this project and aligns with the industry standard for powering most modern consumer electronics. Its ability to efficiently convert electrical power with minimal heat generation makes it ideal for a system like a dance pad, which requires reliable and consistent power delivery to multiple components including sensors, microcontrollers, and LED panels. The compact physical footprint of an SMPS also allows it to be easily integrated into the slim and portable housing typical of arcade-grade or DIY dance pad designs.

Moreover, SMPS units support a wide input voltage range and are capable of providing stable output voltages, which is particularly important in a project where fluctuating power could lead to erratic sensor behavior or missed inputs. While SMPSs do inherently generate electrical noise due to their high-frequency switching, this drawback is mitigated by the inclusion of filtering and decoupling capacitors within the project's custom-designed PCBs. These capacitors help suppress voltage ripple and electromagnetic interference, ensuring the rest of the circuit operates smoothly.

3.4.5 Comparison and Selection

Table 3.9 *Comparison table of the three types of power supplies in the context of the dance pad project's specific needs*

Feature	Unregulated Supply	Linear Regulated Supply	Switch-Mode Supply (SMPS)
Voltage Regulation	Poor	Excellent	Excellent
Efficiency	N/A (Poor)	Very Poor (~30-40%)	Excellent (~80-95%)
Size & Weight	Large & Heavy	Very Large & Very Heavy	Compact & Lightweight
Heat Generation	Moderate	Very High	Low
Output Noise/Ripple	High	Very Low	Low (with proper filtering)
Cost	Low	High (for this power level)	Low to Moderate
Project Suitability	No	No	Yes

Form Factor

Within the SMPS category, there are two main form factors to consider. The first is an open-frame power supply, which is a bare PCB designed to be mounted inside a larger enclosure. While valid, this option requires careful handling to avoid electrical shock and is not ideal for a student project.

The second, more appropriate option is an external enclosed power supply, commonly known as a "power brick." This is the familiar black box used for laptops and other consumer electronics. Its high-voltage components are safely enclosed in a plastic case, and these units come with pre-approved safety certifications like UL and CE, which is a major advantage. It offers a simple, plug-and-play solution that requires no high-voltage wiring by the user. Reputable brands also include excellent internal filtering to minimize noise output.

Final Selection

Based on this analysis, the recommended power supply for the 9-tile dance pad project is a 12V DC, 5A External Enclosed Switch-Mode Power Supply. The 12V output matches the voltage requirement of the IR LED strip, while the 5A current rating provides a safe operating margin above the calculated 3.3A total system load. The external enclosed form factor offers the best combination of safety, convenience, and reliability for this application.

3.4.6 Conclusion and Implementation

The selection of a power supply is a critical design decision that impacts the reliability, safety, and performance of an entire embedded system. After analyzing the technical characteristics of unregulated, linear regulated, and switch-mode power supplies, it is clear that the Switch-Mode Power Supply (SMPS) is the only practical and effective solution for the 9-tile dance pad project.

The project's mixed-load requirement, powering both a high-current LED system and sensitive, low-current microcontrollers, is best served by the high efficiency and excellent regulation of an SMPS. While the potential for electrical noise from an SMPS is a valid concern, this is effectively mitigated by the robust power filtering and decoupling designed into the project's custom PCBs. The ferrite beads, bulk filtering capacitors, and local decoupling capacitors on the Power Hub and Master Controller boards are specifically designed to handle the type of noise an SMPS produces, ensuring clean power for all sensitive components.

The final recommendation is to procure a high-quality, **12V, 5A external enclosed SMPS ("power brick")** from a reputable manufacturer. This choice leverages the benefits of SMPS technology while delegating the complexities and safety concerns of high-voltage AC-to-DC conversion to a pre-certified, off-the-shelf component. This allows the project to focus on its core innovation: the custom PCB design for the modular tile system and the master controller. By making this selection, the project is built upon a foundation of stable, efficient, and safe power, ensuring the best possible chance for a successful outcome.

3.5 Analysis of LED Driver Technologies for High-Power Illumination

3.5.1 Introduction to LED Drivers

Light Emitting Diodes (LEDs) have become the standard for modern lighting applications due to their high efficiency, long lifespan, and fast response times. However, unlike simple incandescent bulbs, LEDs are semiconductor devices that cannot be connected directly to a voltage source. They require a specialized circuit, known as an LED driver, to operate correctly and reliably. The driver's primary function is to regulate the power supplied to the LED, ensuring it receives a consistent forward current to produce stable light output without being damaged.

For the 9-tile dance pad project, the LED driver is a critical component of the computer vision subsystem. It must power a high-current 12V infrared (IR) LED strip that provides consistent illumination for the player-tracking camera. Furthermore, the driver must allow for precise brightness control via Pulse-Width Modulation (PWM) to adapt to varying ambient light conditions, ensuring the computer vision system remains robust and accurate. This paper will explore the fundamental technologies behind LED drivers, compare different circuit topologies, and select the most appropriate design to meet the project's technical and academic requirements.

3.5.2 Fundamentals of LED Operation and Dimming

An LED's brightness is directly proportional to the forward current flowing through it, not the voltage across it. If the voltage is too low, no current flows and no light is produced. If the voltage is too high, the current can increase exponentially, leading to rapid overheating and catastrophic failure. Therefore, the core task of any driver is to provide a constant, controlled current. There are two primary methods for controlling an LED's brightness analog dimming and Pulse-Width Modulation (PWM):

Analog Dimming

This method involves simply reducing the constant current flowing through the LED. While simple, it has significant drawbacks. As the current changes, the color temperature of the LED can shift, which is particularly problematic for applications requiring consistent light quality. It is also less precise and can be inefficient.

Pulse-Width Modulation (PWM) Dimming

This is the superior and more common method. The driver switches the LED on and off at a frequency too high for the human eye or a camera to perceive. The LED is always driven at its optimal forward current, but the duty cycle (the percentage of time the LED is on versus off) is varied. A low duty cycle results in a dim appearance, while a high duty cycle results in a bright appearance. This method maintains a consistent color temperature and allows for very precise, linear brightness control, making it the ideal choice for the project's IR illumination system.

3.5.3 Comparison of LED Driver Circuit Topologies

There are several ways to design a circuit that can drive and dim an LED strip. These range from simple resistive limiters to complex integrated circuits.

Simple Resistor Driver

The most basic approach is to place a resistor in series with the LED strip to limit the current. While functional for single, low-power indicator LEDs, this method is entirely unsuitable for high-power applications. The resistor would need to be very large to dissipate the excess power, generating a massive amount of waste heat. More importantly, it provides no regulation; as the LEDs heat up, their electrical characteristics change, causing the current to drift and the brightness to become unstable.

Linear Constant-Current Driver

A linear driver uses a transistor (like a BJT or MOSFET) operating in its linear region to act as a variable resistor, ensuring a constant current flows to the LEDs. This design provides a very clean, noise-free output. However, like a linear voltage regulator, it is extremely inefficient. It controls the current by burning off all excess power as heat. For the project's ~3A load, a linear driver would become dangerously hot and waste a significant amount of energy, making it an impractical choice.

Simple MOSFET Switch (Low-Side Driver)

This was the initial design considered for the project. An N-Channel MOSFET is used as a simple switch on the low-side (ground connection) of the LED strip. The PWM signal from a microcontroller turns the MOSFET on and off. This circuit is simple, inexpensive, and can handle high currents. However, it provides no current regulation. The brightness of the LEDs will still vary if the main 12V supply voltage fluctuates, and it does not protect the LEDs from potential over-current situations. While functional, it lacks the robustness and precision required for a high-performance system.

3.5.4 Dedicated Switching (Buck) LED Driver IC

The most advanced and professional solution is to use a dedicated Integrated Circuit (IC) designed specifically for driving LEDs. These ICs are complete Switch-Mode Power Supplies (SMPS) on a chip, configured as constant-current buck converters.

How They Work: A buck converter efficiently steps down voltage by switching a MOSFET on and off at a high frequency, using an inductor and diode to smooth the output. An LED driver IC takes this a step further. It includes a feedback loop that monitors the current flowing to the LEDs via a small, external sense resistor. The IC constantly adjusts the switching duty cycle to ensure the output current remains perfectly stable, regardless of changes in input voltage or temperature. These ICs also have a dedicated input pin for a PWM signal, which is used to precisely control the dimming of the constant-current output.

This topology combines the high efficiency of a switching regulator with the precise current control needed for high-power LEDs. It is the most robust, efficient, and feature-rich solution.

3.5.5 Comparison and Selection

A comparison of the viable driver technologies clearly points to the dedicated IC as the superior choice.

Table 3.10 *Comparison table of driver technologies*

Feature	Simple MOSFET Switch	Dedicated Driver IC (e.g., MP24894)
Current Regulation	None	Excellent (Constant Current)
Efficiency	High	Very High
Brightness Stability	Fair (Varies with Voltage)	Excellent
Component Count	Low	High
Design Complexity	Low	High
Protection Features	None	Often built-in (e.g., thermal shutdown)

While a simple MOSFET switch is functional, it does not meet the project's need for precise and stable illumination for the computer vision system. Furthermore, it does not satisfy the academic requirement to integrate a complex reference circuit.

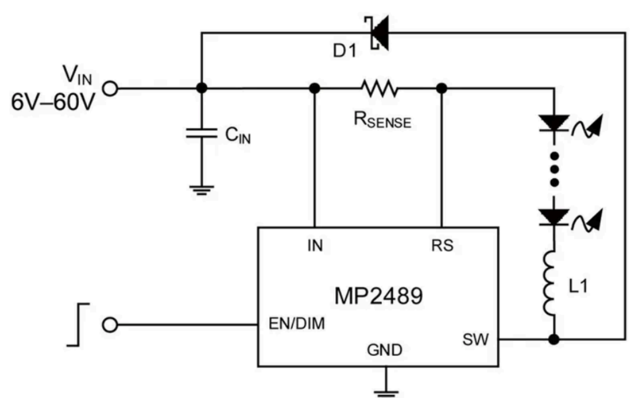


Figure 3.3 *MP2489 LED Driver Circuit Diagram*

The dedicated switching LED driver IC is the optimal solution. It provides the high efficiency and precise constant-current control necessary for reliable performance. The selection of the MP24894 for this project is a direct implementation of this technology. By building the reference circuit for this IC, the project demonstrates a significant level of design complexity, directly addressing the guidelines to "integrate existing reference circuit designs to meet all design requirements." This approach results in a professional-grade, robust, and highly functional LED driver that is perfectly suited to the demands of the dance pad's computer vision system.

3.6 Imaging Subsystem

3.6.1 Monochrome vs RGB Sensor

Selecting the appropriate camera sensor for the S.T.E.P.S vision system was an important decision in the beginning stages of the design process. Technically a RGB or monochrome sensor could be used. Each sensor type has distinct advantages and weaknesses. Monochrome sensors are known for producing high-contrast and high-resolution images due to the absence of color filter arrays.[11] Which can improve clarity and reduce light loss. These sensors are particularly effective in applications where edge detection, structure analysis, or low-light imaging is critical. They also generate smaller data sizes which reduces computational load. These factors are all very promising when considering a major goal of the system is to minimize the computational load and processing time of the vision system.

However, the pose estimation model used in this system is MediaPipe. MediaPipe’s system is optimized for RGB input as it relies on color information to identify and track body landmarks accurately, using patterns in hue and saturation to distinguish between limbs and background.[10] Since RGB sensors capture color images in a format that mirrors human visual perception, these sensors are the best to use for AI models that interpret motion and orientation. In the S.T.E.P.S system, visual feedback is provided to the player via the RGB LED arrays placed in the dance pads. While the motion is captured using the FSR sensors, pose estimation is captured purely through visual tracking. In this case, a monochrome sensor would introduce ambiguity in differentiating the background from the player due to no color differentiation.Therefore, the benefits from using a RGB sensor outweigh those of the monochrome sensors in this application. Despite their slightly higher data rates and lower per-pixel sensitivity. The deficiencies in image contrast or sharpness introduced by the RGB filter array are mitigated through diligent and carefully designed lens system and image scale optimization. This comparison is summarized in table 3.11.

Table 3.11 *RGB vs. Monochrome Sensor Comparison*

Attribute	Monochrome Sensor	RGB Sensor
Image Contrast & Sharpness	Higher (no color filter array)	Lower due to color filter array (CFA)
Light Sensitivity	Higher(no CFA, more light per pixel)	Lower (CFA reduces incoming light)
Data Size	Smaller (single channel)	Larger (3 channels: R, G, B)
Computational Load	Lower (less data to process)	Higher (more data to process)
AI Compatibility	Not supported by MediaPipe (requires RGB input)	Fully supported and optimized for MediaPipe
Background Differentiation	Poor (no color distinction between limbs, background)	Good (color cues help separate limbs and surroundings
Suitability	Not suitable due to MediaPipe incompatibility	Selected for pose tracking and AI model compatibility

3.6.2 Camera Comparison

Several camera modules were considered to balance resolution, frame rate, and price. The chosen camera needed a pixel pitch suitable for resolving ~ 1mm features, a frame rate of at least 30 FPS for smooth gameplay, a cost preferably under $100 to fit system goals, and compatibility with the 850nm illumination. Originally, the Arducam AR0234 USB 2MP global shutter camera was identified as a strong candidate. Its 3.0 µm pixel pitch, RGB output, and use of no IR-cut filter makes it ideal to properly sense the NIR illumination. The global shutter also significantly reduces motion blur, which is important during fast player movement.[29]

Through more research, a lower-cost alternative using the same AR0234 sensor was found through SVPRO. The SVPRO 2MP Global SHutter USB camera module offers similar specifications, including global shutter operation, M12 lens compatibility, and RGB output, but for a better price. Its lower cost makes it more ideal for the prototype phase. However, it may have reduced support and documentation compared to the Arducam-branded product. The Raspberry Pi HQ camera was also considered because of its modular CS-mount compatibility, but its rolling shutter and low frame rate made it less suitable for pose estimation. Overall, the SVPRO AR0234 offers a balanced trade-off between pixel size, cost, and motion capture. The specifications for these candidate cameras are summarized in Table 3.12.

Table 3.12 *Camera comparison table*

Camera	Pixel Pitch(μm)	Frame Rate (FPS)	Price (USD)	Shutter Type
Raspberry Pi HQ (IMX477)	1.55	20	$53.78	Rolling
Arducam AR0234 Global Shutter	3.0	60	$109.99	Global
SVPRO AR0234 Global Shutter	3.0	60	$76.99	Global

3.6.3 Lens Comparison

A wide field of view and short focal length were prioritized to capture the entire 114'' x 114' dance pad area from approximately 1.8m away. Research into wide-angle M12 lenses focussed on models with a horizontal field of view near 93°, focal lengths around 3 mm, minimal distortion, sufficient depth of field, and a price under or as close to $100 as possible.[12] These lenses are well-suited for compact vision systems, and the M12 mount allows easy replacement and adjustment during prototype testing.

Although the SVPRO AR0234 camera module includes pre-installed wide-angle M12 lens, alternative 3 mm M12 aspherical lenses will be evaluated to verify critical requirements such as distortion, IR transmission, and image sharpness for the 850 nm active illumination. This ensures the final lens selection supports consistent MediaPipe landmark detection and meets the project's performance targets. Commercially available aspherical M12 lenses with focal lengths around 3 mm were identified as the best balance of wide-angle coverage and low distortion. Aspherical designs help reduce spherical aberration and maintain sharpness across the entire field of view, which is important for MediaPipe pose tracking and consistent landmark detection.[12] Although a fully custom-designed aspherical lens could in theory optimize every optical parameter, the cost and fabrication complexity made that impractical for this prototype. Therefore, an off the shelf aspherical wide-angle M12 lens will be the baseline option.

After evaluating multiple M12-mount aspherical lenses compatible with the sensor, the Commonlands CIL034 was the most appropriate. This lens has a 3.2 mm focal length with approximately 87° HFOV on the AR0234. This field of view allows the camera to capture the full 114’’ x 114’’ dance pad area from a working distance of 1.8m without excessive mounting height or tilt. Its aspherical design supports reduced geometric distortion, which is beneficial for MediaPipe tracking, while remaining compatible with typical project constraints on cost and mechanical mounting. The depth of field is also not a concern because this lens provides a deep focus range, ensuring the entire player remains sharp throughout motion across the full depth of the dance pad. Two additional lenses were also evaluated for comparison and their specifications are summarized in *Table 3.13*. Both of these lenses provided compatible focal lengths and apertures but did not have the horizontal field of view needed.

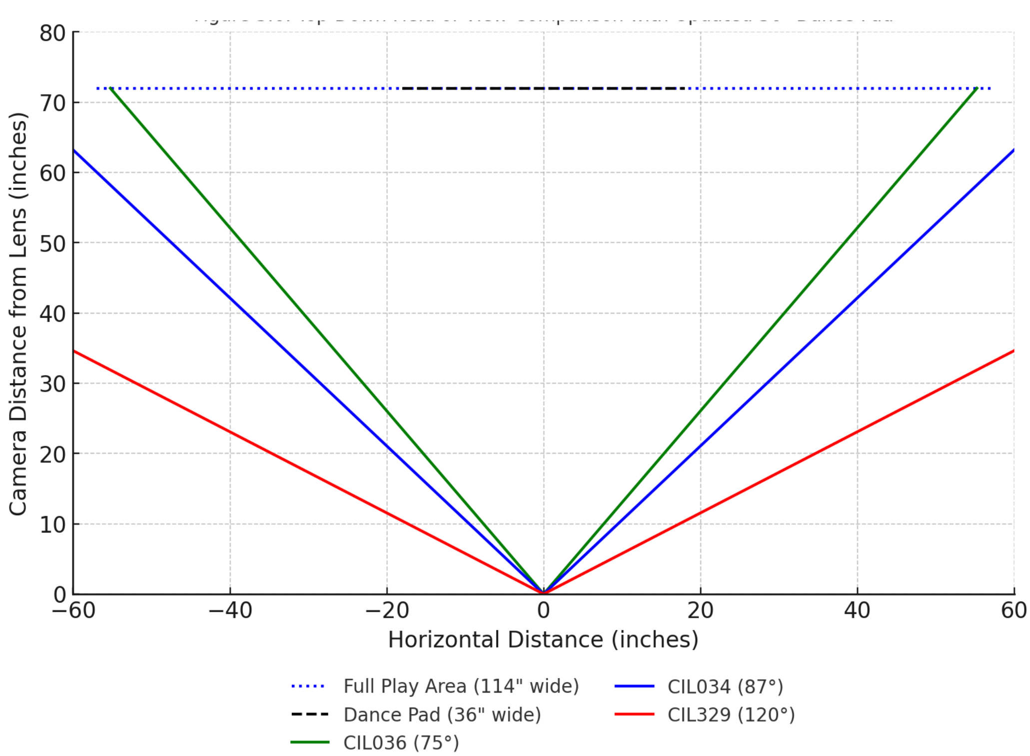


Figure 3.4 *Top-down field of view comparison between candidate M12 lens at 6ft camera mounting distance. The dashed line represents the full 36 inch-wide dance pad area, while the dotted blue line represents the 114" total play area for full-body motion tracking.Wider FOV lenses (e.g., CIL329) cover the full width easily, while narrower lenses like CIL036 may risk cutting off edge activity.*

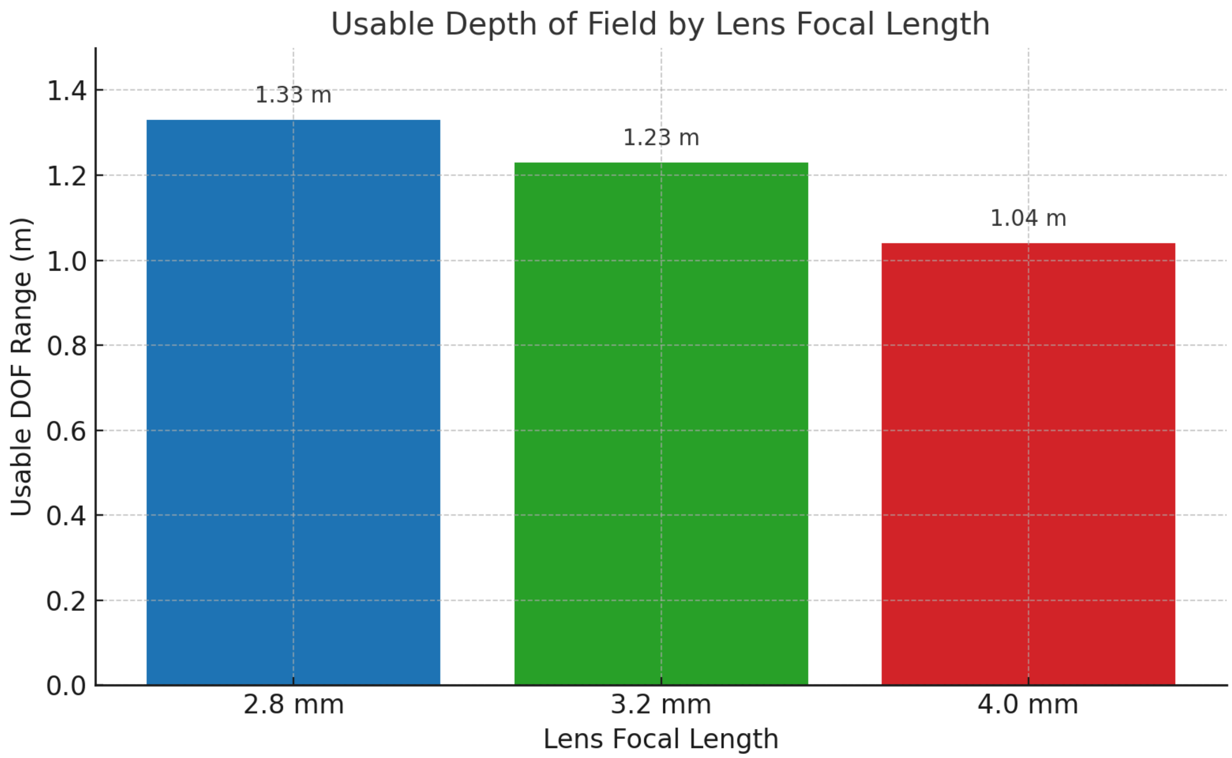


Figure 3.5 *DOF comparison between candidate M12 lenses based on calculated near and far limits at 1.83 m subject distance.*

While the FOV angles of the candidate lenses appear visually exaggerated in top-down diagram in figure 3.4, they are necessary to fully encompass the 2.9 m x 2.9 m player tracking area from a 6 ft mounting distance. Wide-angle lenses, although susceptible to edge distortion, are essential to ensuring full- body pose detection remains accurate without requiring multiple cameras or stitching. Lenses with FOVs ranging from 65 to 120 were considered, with wider-angle options offering increased coverage at the expense of potential barrel distortion near the edges. Despite this, these lenses are well-suited for single-camera full-body tracking, particularly when paired with post-processing or AI algorithms that are resilient to optical distortion. The final selection strikes a balance between angular coverage, pixel resolution, and optical simplicity.

Figure 3.5 compares the usable depth of field range for the 2.8mm, 3.2 mm, and 4.0 mm lenses evaluated for the S.T.E.P.S optical system. As expected, shorter focal lengths provide greater DOF, with the 2.8 mm lens offering the deepest focus range at 1.33 m. The selected 3.2 mm lens maintains a strong DOF of 1.23m, which is sufficient to keep the entire player in focus across the 2.9 m x 2.9 m player tracking area from the 1.8 m camera distance. Although the 4.0 mm lens offers slightly higher image resolution and lower distortion, its narrower DOF of 1.04 m increases the risk of motion blur or focus loss during dynamic movement. Figure 3.5 reinforces the 3.2 mm lens as a balanced choice that preserves focus stability while still meeting resolution and field of view requirements.

Table 3.13 *Lens options and their specifications*

Lens	Focal Length(mm)	Horizontal FOV (°)	Aperture (f/#)	Price (USD)
Commonlands CIL034	3.2	~87	2.4	$39
Commonlands CIL329	2.8	~120	2.4	$39
Commonlands 042	4.0	~68	1.9	$59

Although the selected 3.2 mm lens provided an 87° HFOV, slightly under the calculated 93° requirement, it was deemed acceptable given its low distortion profile, adequate central coverage, and compatibility with the chosen sensor. Further testing will determine if the minimal cutoff at the pad edges will have significant impact on gameplay detection or user experience based on planned camera placement and pose tracking zones during faster/ more dynamic movement .

3.7 LED illumination Technology

For the illumination system, both white & RGB LED strips, as well as individual LEDs were considered. Thorough research went into deciding which option would be the most optimal for this system. While RGB LEDs would offer more visual customization, they raise possible challenges due to their color inconsistencies and power complexity making them less ideal for computer vision tasks. White LEDs would provide better illumination with more uniform color output and higher power efficiency[9], but could also cause discomfort to the players' eyes. Ensuring that MediaPipe is able to consistently perform reliable pose detection is more important than aesthetic lighting, but player comfort is an equally important metric.

With that being said, more research went into finding an alternative that would satisfy both the illumination and user comfort requirements. 850 nm LEDs were found to be the most optimal choice due to their ability to illuminate the player without causing discomfort during game play. 850nm is near-infrared and only partially visible to the player, but it is reliably detected by the camera as illumination.[7] These comparisons are summarized in table 3.15. During the camera selection process, it was important to find a camera that did not include an IR-cut filter to ensure that the 850nm LEDs would be picked up on screen.

The choice between using individual IR LEDs versus LED strips was also considered but LED strips were quickly determined to be the most optimal solution due to their ease of integration and consistent IR output. Standard 12 V LED strips simplify power distribution compared to the more complex constant-current drivers needed for discrete

LEDs, aligning with the system's power efficiency goals. They are also highly practical for testing because they can be easily modified or repositioned as needed. They allow for better uniform and even light coverage across the dance pad area, which supports consistent MediaPipe landmark tracking. While their only downside is having slightly less beam-shaping flexibility, this factor is less important than cost, simplicity and performance, which is why LED strips ultimately outperform discrete individual LEDs in this design.

Table 3.14 *Illumination Method Comparison*

Attribute	RGB LEDs	White LEDs	850nm IR LEDS
Visibility to Player	Fully visible	Bright and visible	Partially or mostly invisible
Comfort / Glare	Moderate (color shifting may distract)	Potential discomfort in dark environment	Comfortable (no glare)
MediaPipe Compatibility	Inconsistent under varied RGB output	High contrast but could saturate camera	Reliable for pose detection
Power Efficiency	Lower (color mixing requires more power	Moderate	High (simple constant voltage)
Ease of Integration	Moderate (requires careful color control)	Easy	Easy 12 V strips
Camera Compatibility	Compatible (RGB input)	Compatible	Compatible (requires no IR-cut filter)
Beam Shaping/ Directionality	Flexible with lenses or domes	Fixed	Slightly less flexible but evenly distributed
Cost and Availability	Moderate to High	Low	Moderate and widely available
Chosen Option for S.T.E.P	Rejected due to inconsistency	Rejected due to glare	Selected for performance and comfort

Table 3.15 Comparison of IR illumination Formats: Individual LEDs vs. LED Strips

Attribute	Individual 850nm IR LEDs	850nm IR LED strips
Integration Complexity	High (requires PCB design, precise alignment)	Low (peel- and-stick, plug-and-play layout)
Driver Circuitry	Requires constant-current sources	Runs off standard 12 V constant-voltage supply
Light uniformity	Potential hotspots, difficult to evenly space	Even illumination across play area
Reusability/ Testing	Difficult to reconfigure during prototyping	Easily repositioned or cut to fit
Beam Shaping Control	Higher flexibility with lenses or reflectors	Lower flexibility. Fixed emission angle
Cost and availability	Often lower per LED but adds up with drivers and PCB work	Cost-effective for large areas widely available
Chosen Format for S.T.E.P	Rejected due to complexity and inconsistency	Selected for simplicity, uniformity, and speed of setup

3.7.1 LED Part Comparison

The main selection criteria for these strips was their ability to provide sufficient radiant flux to support reliable MediaPipe detection, a wide beam angle near 120°for consistent coverage, and moderate power consumption to fit within the available 12 V supply constraints. Three candidates were compared and their specifications were summarized in Table 3.16. The Waveform IRFlex 850nm, DC12/24V 5050 SMD strip, and the 360 Digital signature 3528 SMD strip. The 5050 SMD option provides both high output and a moderate price point for 60 LEDs per meter and a power draw of approximately 14.4 W/m. 3528 SMD offers lower power consumption but has a higher price per reel making it less ideal. The 2835 SMD strip provides the most LED density at 120 LED/m and has the lowest power draw of about 9.6W/m, making it the most premium candidate of the three at $55 per 5m.

All three options maintain a wide beam angle near 120°, supporting robust coverage of the player's body during movement. Based on these tradeoffs such as cost and power, the 3528 SMD strip was selected as the baseline for initial prototyping. This is due to its cost balance, radiant power, and easy 12 V integration with time-multiplexed control zones. Final LED strip selection will be confirmed after prototype testing of illumination uniformity and camera sensitivity.

Table 3.16 *LED Strip comparison table*

Feature	Waveform IRFlex 850nm	DC12/24V	360 Digital Signage
LED Type	2835 SMD	5050 SMD	3528 SMD
Beam Angle	120	120	120
LED Density (/m)	120	60	60
Power(W/m)	9.6	14.4	7.2
Price ($)	$55	$28.98	$45.89

3.8 Embedded System Development Languages

Our system will require software development both for the microcontroller unit (MCU) and the rhythm game interface. In this section, we evaluate the most suitable programming languages for each component based on criteria such as speed, ease of development, memory control, and hardware access. The goal is to identify the best language for programming our MCU and, if necessary, a separate language best suited for developing the rhythm game.

3.8.1 C

C is a low-level language most commonly used for programming MCUs. It provides direct access to hardware while offering a cleaner abstraction than assembly language. Because C compiles directly into machine code with minimal runtime overhead, it allows extremely fast execution and real-time responsiveness, which is critical for embedded applications like sensor reading or LED control.

Unlike high-level languages, C has no garbage collection, safety checks, or virtual machines. Instead, the programmer is responsible for memory allocation and hardware interfacing. While this increases development complexity, it allows precise control over peripherals like GPIOs, ADCs, and timers. C is widely supported across virtually all microcontroller platforms, including Teensy, Arduino, STM32, and ESP32. Overall, C is not the easiest to learn, but it is the most efficient and reliable language for embedded development.

3.8.2 Python

Python is a high-level, interpreted language known for its simplicity, readability, and rapid development. While it is not suitable for low-level embedded programming (due to memory and speed limitations), it is commonly used for PC-side applications, prototyping, and game development, especially when paired with frameworks like Pygame for rhythm games.

In embedded systems, Python is occasionally used on SBCs like the Raspberry Pi, but not on MCUs like Teensy, where C or C++ are more appropriate. However, Python can be used to simulate or visualize data from the dance pad or build a simple game interface that responds to USB HID inputs from the MCU. Its vast ecosystem and ease of integration with USB devices make it a strong candidate for the rhythm game software, but not the microcontroller firmware.

3.8.3 Java

Java is another high-level language often used in application development, especially for cross-platform environments. Like Python, Java is not ideal for embedded MCU programming due to its reliance on the Java Virtual Machine (JVM), which adds significant overhead and lacks the direct hardware access needed for real-time control.

However, Java can be used effectively for developing desktop-based rhythm games or user interfaces that communicate with the dance pad via USB. Java's strong support for event-driven programming, GUI libraries (like JavaFX or Swing), and device input handling makes it a reasonable choice if the rhythm game requires more structure or modularity than Python might provide. Its portability and object-oriented architecture may be overkill for simple games but could benefit larger or more scalable systems.

3.8.4 Embedded System Development Languages Selection

We are using C language to program our Arduino Leonardo microcontroller because it offers the most efficient, direct, and low-overhead way to interact with the hardware components of our system, specifically the FSRs, RGB LEDs, and USB HID interface. C provides precise control over memory, peripherals, and timing, which is crucial for our real-time step detection and LED response requirements. Unlike higher-level languages, C allows us to write low-latency code that can directly manipulate GPIO pins, read analog signals from FSRs via ADC channels, and update LED states with minimal delay.

Arduino Leonardo supports C (and C++) natively through the Arduino IDE, allowing us to easily configure the board for USB HID communication, which is essential for translating foot inputs into instant keyboard-like signals recognized by the game. Since our system is latency-sensitive and must respond with near-instant feedback for both impact and press-and-hold inputs, using C gives us the deterministic performance we need. Overall, C is the most appropriate language for programming the embedded side of our dance pad system, offering speed, reliability, and full access to the hardware's capabilities.

Table 3.17 *Top development languages for MCU comparison table*

Feature	C	Python	Java
Speed & Performance	excellent (low overhead)	poor for embedded, decent for PC apps	moderate (JVM overhead)
Ease of Development	moderate	very high	high
Memory Control	manual	automatic	automatic
Hardware Access	direct access	limited	indirect
Best Suited For	embedded systems	PC-side prototyping, rhythm game logic	scalable PC applications, GUI

3.9 Computer Vision

3.9.1 Pose Estimation Techniques

3.9.1.1 Classical Techniques

Before the widespread adoption of deep learning, pose estimation was achieved through traditional computer vision methods, many of which relied on handcrafted features and rule-based pipelines. These classical techniques were computationally efficient and suitable for the hardware available at the time, but they came with strict environmental limitations and a lack of generalization across diverse scenes.

One of the earliest and most intuitive techniques used in classical pose estimation was background subtraction. This method attempts to isolate the moving subject from a static background by identifying changes in color or brightness between frames. Typically, a “background model” is captured or assumed, and then each new frame is compared pixel-by-pixel to detect motion. Basic versions used simple frame differencing, while more advanced versions employed Gaussian Mixture Models (GMMs) to handle subtle lighting fluctuations or gradual background changes. In a constrained environment this method could effectively identify silhouettes or regions of interest, forming the basis for further analysis like gesture recognition or contour tracking. However, background subtraction quickly broke down in real-world scenarios. Environmental noise such as flickering lights, moving shadows, or dynamic backgrounds like waving trees introduced errors. Furthermore, it required the camera to be stationary, as even slight camera movement could disrupt the entire model. While it was a useful tool for early motion tracking experiments, its fragility under real conditions made it impractical for generalized use, especially in games or public environments.

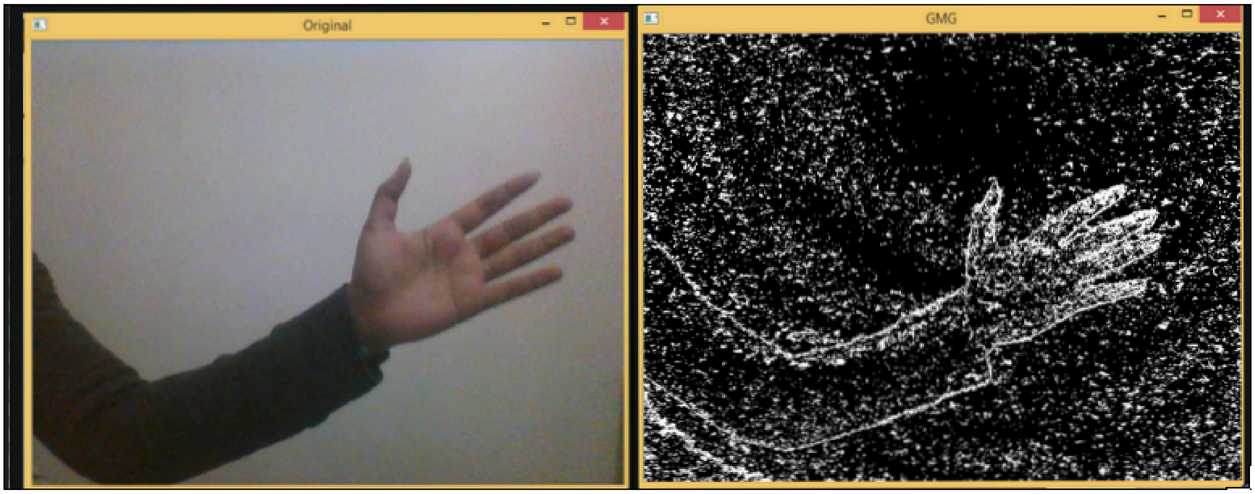


Figure 3.6 *Example of Background Subtraction with OpenCV*

Another classical technique that provided more temporal insight into motion was optical flow. Instead of comparing single frames against a background, optical flow analyzes the apparent movement of pixels between consecutive frames to estimate motion vectors. This technique is especially useful for capturing the direction and velocity of motion, making it ideal for analyzing limb movements or detecting walking and waving gestures. The two primary approaches being dense flow and sparse flow offered different trade-offs between resolution and performance. Optical flow has been used in robotics and video analysis to infer actions from continuous motion, and in some early gesture-controlled systems. However, it came with its own set of limitations. Large or fast movements often led to errors due to motion blur or discontinuities, and occlusion, when one part of the body hides another, posed a major challenge. Moreover, flow estimation becomes ambiguous in areas with uniform texture or lighting, where pixel intensity doesn't change enough to provide directional clues. In the context of multi-person scenes or complex poses, optical flow was insufficient on its own, often requiring additional heuristics or model-based constraints to be effective.

Contour detection and shape matching techniques sought to extract body outlines by identifying edges and fitting known templates or convex hulls. These methods enabled rough body pose estimation but failed in the presence of overlapping limbs, varied clothing, or non-standard postures. Similarly, pose-from-silhouette techniques relied on the shape of a segmented figure to infer likely joint locations, using statistical models of human anatomy. These systems performed reasonably well for frontal or side views but were limited by the inherent ambiguity of silhouettes in 2D projections.

Among classical methods, pictorial structures marked a significant conceptual shift by introducing a more structured way to represent human pose. Rather than analyzing motion or edges in isolation, they use a computer science graph-like representation where nodes correspond to body joints and edges represent limbs. Each part was associated with a probability distribution describing its likely appearance, and the spatial relationships between parts were encoded through geometric constraints. The system would then search for the configuration that maximized the overall likelihood across the image, essentially solving an optimization problem to infer the most plausible pose. This approach was robust to some extent against partial occlusion and varying camera angles, making it more suitable than silhouette or contour-based methods for challenging environments. However, it came at a cost. The inference step was computationally expensive, especially for full-body models with many degrees of freedom. These systems also relied heavily on strong contrast and clean segmentation to reliably detect features in the first place. As a result, while pictorial structures provided a powerful theoretical foundation for understanding pose, their practical use was limited until more efficient algorithms and stronger feature representations emerged in the deep learning era.

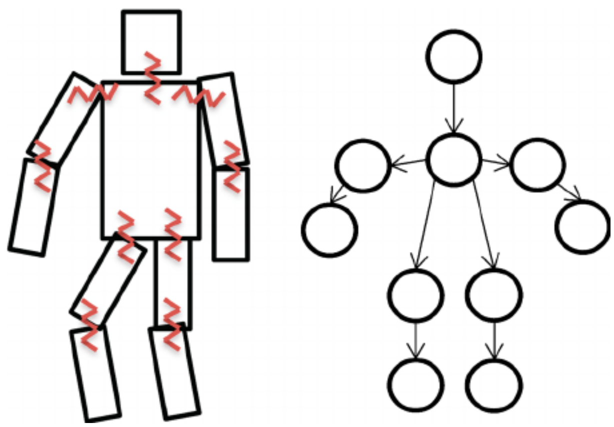


Figure 3.7 *Graph representation of a typical pose for a human body*

A major step forward came with the introduction of the Microsoft Kinect in 2010, which helped bridge the gap between classical vision pipelines and modern learning-based systems. The Kinect utilized structured infrared light to capture depth data in addition to standard RGB imaging. This enabled accurate 3D skeletal tracking without requiring wearable sensors or markers. Microsoft's implementation used machine learning decision forests trained to identify joint positions from depth images, allowing for real-time, multi-user tracking on modest consumer hardware such as the Xbox 360. Though highly influential, Kinect's depth-based approach suffered from hardware limitations, including limited range, sunlight interference, and platform dependency.

Despite their constraints, these classical techniques laid the groundwork for the learning-based methods that followed. They demonstrated the feasibility of markerless tracking and inspired early applications in gaming, gesture control, and human-computer interaction. The lessons learned from these systems informed the design of modern neural network architectures and dataset collection strategies used in today's pose estimation models.

3.9.2 Modern Techniques

As hardware improved and large annotated datasets became available, convolutional neural networks (CNNs) emerged as the dominant paradigm for pose estimation. These models replaced handcrafted pipelines with data-driven learning, enabling greater robustness, scalability, and accuracy.

The emergence of CNNs has drastically transformed the field of human pose estimation, enabling high-accuracy, real-time detection of key body joints from standard RGB imagery. Unlike classical techniques that relied on handcrafted features or background modeling, modern CNN-based systems can generalize to a wide variety of poses, environments, and occlusion conditions through extensive training on large-scale annotated datasets.

One of the most influential convolutional neural network-based pose estimation frameworks is OpenPose[19], developed by the Carnegie Mellon Perceptual Computing Lab. OpenPose was the first open-source system capable of performing real-time, multi-person 2D pose estimation with relatively high accuracy. It uses a method called Part Affinity Fields, which predicts not only the positions of individual body key points such as wrists, elbows, and knees, but also the directional associations between these keypoints. This allows the system to effectively group detected joints into separate individuals, even when multiple people are present in the same scene or overlapping in view.

While OpenPose set a major milestone in pose estimation research, its accuracy comes with significant computational demands. It typically requires a dedicated GPU in order to achieve real-time performance, and its performance degrades considerably on devices without powerful hardware. As a result, OpenPose is not well-suited for mobile, embedded, or browser-based environments where memory, power, and processing capabilities are limited. The framework is built on Caffe and makes extensive use of OpenCV for tasks such as image preprocessing, rendering keypoints, drawing skeletal lines, and managing post processing routines. This design makes it modular and relatively easy to modify, but also heavy and resource-intensive compared to more modern, lightweight alternatives.

In practice, many of the tasks handled by OpenPose's post processing pipeline can be re-implemented with simpler code using OpenCV and basic geometric rules. For example, drawing connections between keypoints based on distance thresholds or grouping joints into a rough pose can be achieved using a few lines of logic, especially when used in combination with another library like MediaPipe to handle keypoint detection. This makes OpenPose seem unnecessarily complex for applications that only require basic gesture detection, such as identifying if someone is raising their hands, jumping, or standing still.

Figure 3.8 *OpenPose working on a large amount of subjects*

In contrast to OpenPose, MediaPipe[21] Pose was developed by Google with an emphasis on speed, efficiency, and platform flexibility. It was designed to operate on low-powered devices without sacrificing too much accuracy. The system uses a two-stage convolutional neural network pipeline. First, it detects the region of interest around a person using a lightweight detector. Then, it applies a landmark model to predict thirty-three key points across the entire body, including finer details such as fingers, feet, and facial landmarks.

What sets MediaPipe Pose apart is its ability to deliver real-time performance even on CPUs and embedded systems. It runs reliably on mobile phones, Raspberry Pi boards, and NVIDIA Jetson devices without the need for a dedicated GPU. Its modular architecture is built around a graph-based processing model, where each component in the pipeline is represented as a node. This allows developers to customize, replace, or extend parts of the pipeline, such as input normalization, inference logic, or output smoothing.

In addition to its lightweight design, MediaPipe includes smoothing filters and motion tracking features that reduce jitter and increase temporal stability. This makes it especially suitable for applications involving fast movements or inconsistent lighting. Although it may not match OpenPose in multi-person detection or sub-pixel keypoint accuracy, its practical balance between speed, resource usage, and flexibility makes it an ideal choice for mobile games, real-time interaction, and low-latency gesture recognition.

OpenPose remains a powerful benchmark in academic and industrial settings where multi-person detection, fine-grained keypoint accuracy, and research flexibility are prioritized over efficiency. Its ability to consistently detect multiple overlapping subjects and produce detailed skeletons makes it valuable for high-end applications such as motion capture, live broadcasting, and scientific analysis. However, its reliance on GPU acceleration, heavy memory footprint, and large model size make it impractical for use in resource-constrained environments, such as mobile games or embedded systems. MediaPipe Pose offers a more balanced solution by maintaining reasonably high accuracy while optimizing for speed, modularity, and ease of deployment. Its ability to operate smoothly on CPU-based systems allows developers to use it in real-time applications where latency must be minimized and where external hardware acceleration is not feasible. Furthermore, its graph-based architecture and wide platform support (including Android, iOS, and desktop) enable developers to prototype and iterate across a broad range of devices with minimal changes to the underlying codebase.

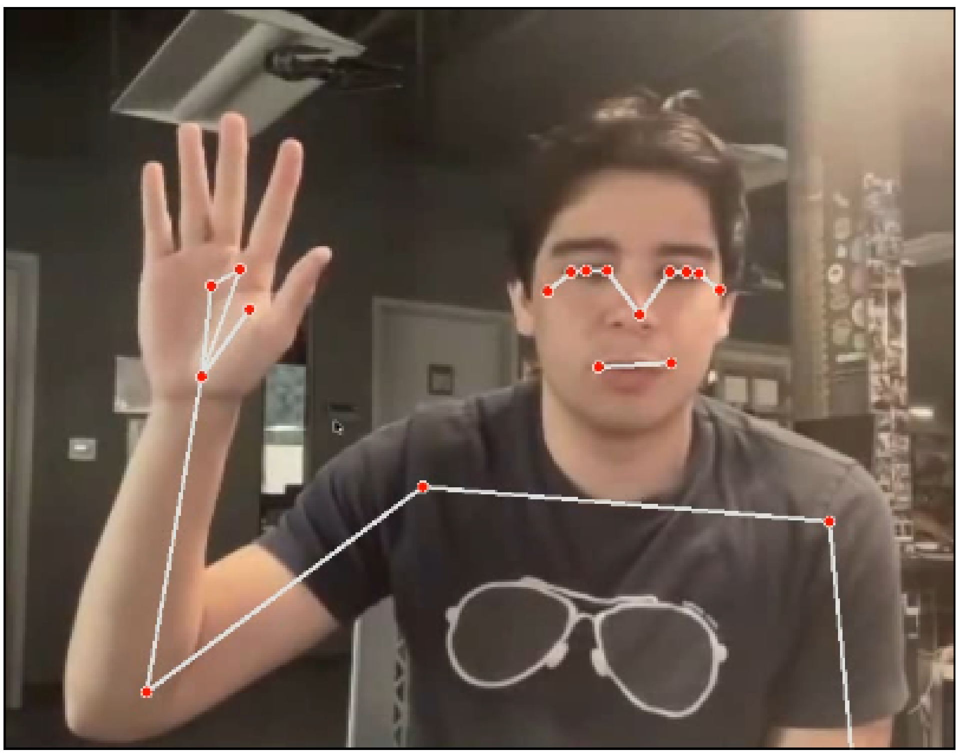


Figure 3.9 *Example prototyping of OpenCV and MediaPipe on Christopher Solanilla*

For scenarios demanding ultra-fast performance and even lighter computational demands, BlazePose[20] takes this optimization a step further. Its use of single-shot detection eliminates the need for separate region proposals or sequential processing steps, significantly reducing latency. Combined with model quantization and neural architecture search, BlazePose achieves performance that exceeds 30 frames per second on modern smartphones using only CPU resources. This makes it highly suitable for applications in fitness tracking, augmented reality (AR), and gesture-based interfaces where responsiveness and battery efficiency are critical. Additionally, its prediction of 33 keypoints, including subtle facial and foot positions, enables more expressive pose understanding despite its lightweight design. The most intriguing feature of BlazePose is that it is able to predict within a high accuracy x, y, and z coordinates for problems that may require 3D location.

In the context of our rhythm-based game, where low latency, reliable body tracking, and smooth animations are essential for gameplay responsiveness, both MediaPipe Pose and BlazePose represent ideal choices. Given that the game must run on commodity hardware without access to high-end GPUs, the selection of a lightweight model is not only a technical preference but a hard requirement. MediaPipe's modular pipeline offers a flexible starting point for tuning the pipeline to match the rhythm game's input structure and timing constraints, while BlazePose introduces an opportunity to push performance boundaries even further if frame rate becomes a bottleneck. These models align well with the constraints of real-time play and allow us to maintain a consistent experience across platforms, from desktops to embedded consoles.

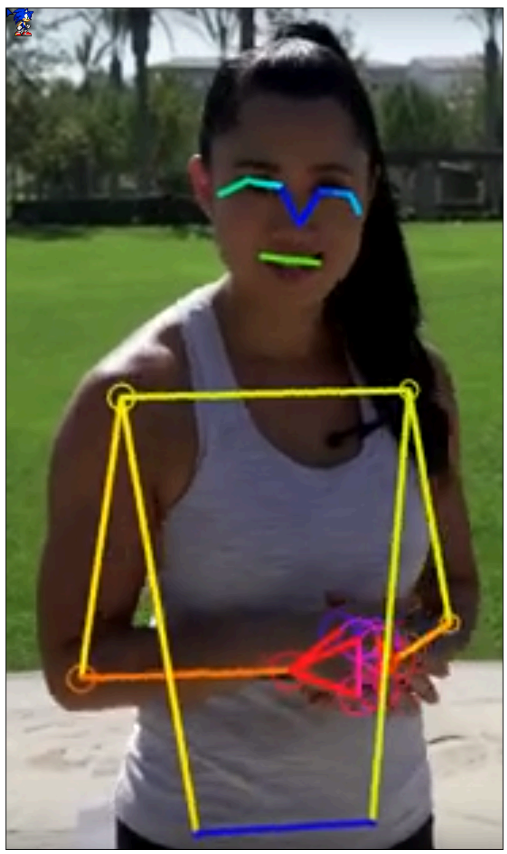


Figure 3.10 *BlazePose demonstrating high performant pose tracking capabilities with 3 dimensions*

3.9.3 Computer Vision Library Selection

After evaluating several options for implementing pose detection and real-time vision-based input, we have selected MediaPipe and OpenCV as the primary libraries for our computer vision subsystem. This decision reflects both technical capabilities and our prior development experience.

MediaPipe, developed by Google, offers a lightweight and efficient pose estimation pipeline that can run on commodity hardware, including embedded systems such as the Jetson Nano. It supports real-time inference with acceptable latency, making it suitable for our rhythm game's timing-sensitive requirements. MediaPipe provides prebuilt models such as BlazePose, which can detect and track 33 body landmarks with a balance of accuracy and speed. Its modular graph-based architecture allows customization of the processing pipeline, enabling us to skip unused stages and fine-tune performance parameters as needed.

OpenCV complements MediaPipe by offering a wide range of image processing utilities, camera input handling, and matrix operations that are useful for gesture classification and data smoothing. It also provides tools for visualization, geometric transformations, and real-time filtering, all of which are valuable during debugging and refinement of the pose recognition system.

Our previous work with these libraries in related projects, such as a pose-driven accessibility tool, has shown that they are reliable, portable, and efficient when compiled natively in C++ using Bazel. By combining MediaPipe's pose estimation models with OpenCV's low-level processing capabilities, we avoid the need to train models from scratch and instead benefit from a well-supported and mature ecosystem.

Table 3.18 *Comparison table of different computer vision technology*

Tool	Speed	Accuracy	Hardware Requirement	Multi-person Support	Best Use Case
OpenCV	Fast	Low to Medium	Works on any CPU	No	Basic gesture logic, simple demos
OpenPose	Slow	High	Requires powerful GPU	Yes	Research, motion capture

MediaPipe	Fast	Medium to High	Runs on CPU/mobile	No	Mobile apps, real-time interaction
BlazePose	Very Fast	Medium	Optimized for CPUs	No	Fitness apps, AR, fast input

3.9.4 Prototyping with MediaPipe and OpenCV

To better understand the practical limitations and advantages of MediaPipe Pose, early prototyping work was conducted prior to this project using a custom-built C++ application developed for a separate accessibility-focused project. This project, titled *Body Language Detector*, was created as part of a hackathon hosted by my workplace. Its goal was to explore the use of computer vision in assisting visually impaired users by interpreting body language in real time. The application used a webcam feed to identify and analyze nonverbal cues, such as hand gestures and body posture, and translate these visual signals into auditory or textual feedback. The project aimed to give blind or low-vision users more access to subtle social signals that are typically communicated without speech.

To meet the performance requirements of real-time interaction, the application was developed in C++ and used GTK for the graphical interface. This allowed for a native, low-overhead UI that could display pose information and system feedback without introducing latency. OpenCV was used to handle image processing and visualization tasks, while MediaPipe Pose provided the backbone for keypoint detection. Rather than relying on Python, which would have simplified development but introduced performance overhead, the entire pipeline was compiled using Bazel to ensure smooth operation on Linux systems. Particular care was taken to support CPU-only inference, making the program compatible with resource-constrained environments such as embedded systems or older computers. The system was tested on a Gentoo-based Linux setup, demonstrating stable performance and reliable keypoint detection without the need for external GPU acceleration.

As seen in Figure 3.9, the prototype successfully tracked the user's full-body pose using a standard webcam. Landmarks such as the shoulders, elbows, knees, and wrists were recognized in real time, and gesture logic was applied by calculating the relative distances and angles between these points. For example, the application could detect whether a person had raised both arms, crossed their arms, or shifted their weight to one side. These features allowed for basic pose classification and helped shape an understanding of what kinds of gestures could be reliably detected with lightweight pose estimation frameworks. The experience of building gesture classifiers using simple geometric heuristics informed the rhythm game's input design, where fast, unambiguous pose recognition is essential for gameplay accuracy and responsiveness.

The project is available on a public GitHub repository and includes build instructions, MediaPipe submodules, and example code for evaluating gestures using OpenCV and landmark data. Although the original purpose of the tool was to improve accessibility for users with vision impairments, the prototyping process provided valuable insights into the deployment of real-time pose estimation systems on constrained hardware. Specifically, it helped highlight the trade-offs between model complexity and responsiveness, and demonstrated the effectiveness of MediaPipe's filters and tracking logic in noisy or unpredictable environments. These lessons directly informed the technical direction of the project. Both offer reliable, low-latency pose estimation that is well-suited for interactive applications running on systems without high-end graphics hardware.

3.9.5 Considerations for Embedded Systems

Although the primary development and testing of the rhythm game is conducted on desktop-class hardware such as gaming laptops or PCs, a long-term stretch goal of this project is to deploy the system on an embedded platform such as the NVIDIA Jetson Nano. This would allow the game to run as a standalone unit with integrated computer vision, enabling a more portable and self-contained experience. While not required for the core project deliverables, the embedded deployment scenario has been considered throughout the design process to ensure future compatibility and ease of migration. Achieving this goal introduces a number of technical constraints, particularly related to computing power, memory, and thermal management that must be accounted for in both the game engine and pose estimation pipeline.

3.9.5.1 Hardware Capabilities of the Jetson Nano

The NVIDIA Jetson Nano serves as a compact and cost-effective embedded computing platform aimed at enabling edge AI applications. It is built around a quad-core ARM Cortex-A57 CPU operating at up to 1.43 GHz, paired with a 128-core Maxwell GPU capable of executing parallel computations required by deep learning inference. The board is equipped with 4 GB of LPDDR4 memory, which is shared between the CPU and GPU, making memory contention a critical factor in system performance. This unified memory architecture reduces the overhead of copying data between processors but also introduces constraints when both the vision pipeline and the game engine are competing for memory bandwidth and capacity.

Despite its modest specifications compared to desktop-class GPUs or higher-tier Jetson modules such as the TX2 or Xavier NX, the Nano is capable of running real-time inference workloads when models are properly optimized. However, it cannot support large-scale convolutional neural networks without suffering from memory exhaustion, cache contention, or reduced throughput. Pose estimation systems that rely on deep neural networks, particularly those involving multiple stacked layers or high-resolution heatmaps, can quickly push the Jetson Nano to its limits. OpenPose, for example, typically requires over 2 GB of RAM just for model inference at full resolution, and relies on multi-branch processing stages to estimate keypoints and limb associations. These demands make it infeasible to deploy OpenPose in its standard form without significant pruning, quantization, and architectural simplification. Even in highly constrained configurations, OpenPose achieves only 3 to 5 frames per second on the Jetson Nano, rendering it unsuitable for real-time interactive use in latency-sensitive applications like rhythm games.

In contrast, lightweight alternatives such as MediaPipe's BlazePose or Pose models are specifically designed to operate under embedded constraints. MediaPipe relies on a streamlined architecture that detects a single person per frame using a detector-tracker approach, typically involving an initial region-of-interest (ROI) proposal followed by a landmark regression stage. These stages are implemented as modular nodes within a dataflow graph that can be reconfigured or pruned depending on the use case. By reducing input resolution, disabling optional filters, and leveraging fixed-point arithmetic, the MediaPipe pipeline can be tuned to achieve 15 to 25 frames per second on the Jetson Nano, even when running alongside other processes such as camera capture and basic game logic.

Furthermore, the Jetson Nano’s GPU can accelerate some portions of the pipeline through CUDA, though MediaPipe does not natively integrate with NVIDIA TensorRT, which limits the ability to fully offload inference from the CPU unless custom GPU kernels are developed. Nevertheless, the combination of a streamlined pose estimation model, configurable processing graph, and low-memory footprint makes MediaPipe a viable candidate for real-time single-user gesture recognition on this hardware platform.

3.9.5.2 Software Stack and Runtime Optimization

Computer vision applications on the Jetson Nano are typically developed using the NVIDIA JetPack SDK, which includes CUDA for GPU computation, cuDNN for deep learning operations, and TensorRT for inference optimization. While these tools are powerful, they are not directly compatible with every pose estimation framework. MediaPipe, for example, is not natively integrated with TensorRT and instead relies primarily on CPU inference or custom GPU pipelines. This can restrict its ability to fully leverage the Nano's GPU unless a custom build is created and compiled specifically for Jetson’s architecture. In contrast, OpenPose requires significant manual configuration to function on the Jetson Nano. This includes aligning CUDA versions, recompiling OpenCV with GPU support, and configuring swap space to avoid out-of-memory crashes. These dependencies and the resulting fragility of the system make OpenPose impractical for a rhythm game targeting consistent frame rates and predictable behavior.

MediaPipe provides a more stable and lightweight alternative. It offers precompiled binaries for both Python and C++ that support CPU inference, and its modular graph-based processing structure allows developers to disable optional components to reduce resource consumption. When paired with inference backends such as TensorFlow Lite or the ONNX Runtime, MediaPipe can run efficiently on the Jetson

Nano with minimal adjustments. The flexibility of this architecture is particularly advantageous when attempting to tailor the pose detection pipeline to balance latency, precision, and power usage.

3.9.5.3 Thermal and Power Management Constraints

In embedded systems development, particularly those involving computer vision workloads, power consumption and heat generation become critical engineering concerns that must be addressed early in the design process. The NVIDIA Jetson Nano provides two selectable power modes: a default 5-watt mode and a more performance-oriented 10-watt mode, which requires a barrel-jack power supply for stable operation. While enabling the higher power mode improves the responsiveness of both neural network inference and general system tasks, it also introduces higher thermal output, which can become problematic in the absence of sufficient cooling mechanisms. Continuous execution of convolutional neural network models, such as those used for pose estimation, alongside a concurrently running rhythm game engine, places sustained computational stress on the device, increasing the risk of thermal throttling if the system exceeds its safe temperature thresholds.

To mitigate thermal buildup and avoid performance degradation over time, the Jetson Nano requires active cooling solutions such as compact fans or aluminum heatsinks. Even with such additions, software design must complement the hardware limitations by adopting strategies that minimize unnecessary computational overhead. For instance, reducing the frequency of camera frame polling or selectively enabling vision processing only during gameplay segments that require it can dramatically lower heat output. The rhythm game being developed does not rely on continuous full-body pose monitoring at all times; instead, it only requires pose classification during specific gameplay prompts that challenge the player to perform a particular movement. This creates an opportunity to disable or suspend the pose estimation system during periods when it is not in use, thereby reducing CPU and GPU workload and allowing the device to operate in a lower-power state for the majority of gameplay.

Such an approach aligns well with the real-time demands of rhythm games, where pose detection can be activated just prior to a prompt and then evaluated in a short time window before being deactivated again. This intermittent activation pattern not only supports power savings but also simplifies thermal management, since the vision pipeline is no longer running continuously in the background. Moreover, additional efficiency can be gained by limiting the input resolution of the camera, lowering the target frame rate for pose estimation to around 15 frames per second, or batching non-time-critical processing tasks so they execute during quieter moments of gameplay. These strategies allow for more predictable thermal behavior and reduce the likelihood of the device reaching critical temperatures that trigger throttling or system instability

3.9.5.4 Game Engine Integration and Resource Synchronization

The rhythm game is being developed using the Godot Engine, which introduces additional integration considerations. Godot allows for C++ extensions through

GDExtension or GDNative, making it possible to incorporate MediaPipe's C++ API directly into the game. However, careful synchronization between the engine and the pose estimation system is critical. Since both the game and the pose detector may require access to OpenGL contexts or video memory, concurrent usage can result in resource contention or graphical glitches if not managed properly. To avoid such conflicts, pose estimation should be executed in a dedicated thread or separate process. Pose data can then be shared with the game engine using inter-process communication or memory-mapped files, ensuring that Godot can operate smoothly without interruptions caused by the vision pipeline.

This architectural decoupling also helps to minimize the impact of intermittent performance drops in the vision system. Since pose detection and gameplay operate semi-independently, short delays in processing frames do not immediately disrupt game logic. Instead, previously detected poses can be held until the next update cycle, maintaining consistency for gameplay elements that rely on accurate gesture input.

3.9.5.5 Memory Footprint and Latency Targets

The Jetson Nano’s 4 GB of RAM must support the operating system, the game engine, camera buffering, model inference, and other runtime operations. MediaPipe’s pose detection typically consumes between 100 and 200 megabytes of memory, depending on image resolution and whether smoothing filters are enabled. Additional RAM is used by the video capture pipeline, graphical assets, and internal Godot processes. Without proper profiling, it is easy to overcommit memory and cause performance to degrade unpredictably.

For rhythm-based gameplay, responsiveness is crucial. The full pipeline from camera input to pose interpretation and game response should ideally remain under 150 milliseconds of latency. Delays beyond this threshold can disrupt the timing-sensitive nature of rhythm gameplay and break player immersion. Achieving this target requires efficient memory management, parallelization of processing stages, and reduction of unnecessary overhead in both vision and game logic subsystems.

3.9.5.6 Summary and Design Tradeoffs

Based on the platform analysis, MediaPipe remains the most viable pose estimation solution for the NVIDIA Jetson Nano. It balances performance, configurability, and platform compatibility in a way that suits the needs of embedded game development. While OpenPose offers more detailed multi-person tracking, its memory requirements and complex setup disqualify it from practical use in this environment. BlazePose, as an extension of the MediaPipe ecosystem, provides an additional fallback option in cases where even greater inference speed is necessary or if resource usage exceeds acceptable thresholds. By carefully managing resource allocation, thermal conditions, and runtime behavior, it is possible to build a responsive and reliable rhythm game experience that leverages pose estimation on embedded hardware without sacrificing performance.

3.9.6 Programming Languages for Game and Computer Vision

The selection of programming languages for this project plays a central role in shaping both the development workflow and the real-time performance of the computer vision and game engine components. Given the nature of embedded systems and the integration of computer vision models for pose detection, careful language choices must be made that reflect both the technical demands of the system and the development constraints faced by the team.

For the computer vision subsystem, both C/C++ and Python are widely adopted in the field and are supported by key libraries such as OpenCV and MediaPipe. Python offers a fast and accessible prototyping experience due to its dynamic typing, extensive library ecosystem, and concise syntax. MediaPipe, in particular, provides precompiled Python wheels that allow for rapid experimentation and integration with OpenCV, making it an ideal language for early-stage development and algorithm testing. However, Python's interpreted nature and higher runtime overhead make it less suitable for performance-critical or resource-constrained deployments, especially on embedded hardware like the Jetson Nano.

In contrast, C/C++ offers significant advantages in terms of execution speed, memory control, and portability, particularly for embedded and production environments. MediaPipe's core is written in C++ using the Bazel build system, which provides high performance but also introduces greater complexity during compilation and integration. The C/C++ API grants lower-level access to graph construction, buffer management, and custom kernel development, enabling developers to fine-tune pipelines for optimal performance. However, setting up a MediaPipe C/C++ environment on Linux, especially for cross-compilation or GPU acceleration, can be challenging and may require in-depth knowledge of build systems and dependency management.

Given the tradeoffs between flexibility and performance, many projects begin with Python for rapid prototyping before transitioning to C++ for deployment. However, due to prior experience working with MediaPipe, OpenCV, and embedded C++ development, we are able to bypass the initial Python prototyping phase and move directly into a C++ implementation. This decision allows us to focus our efforts on building a production-ready system from the outset while maintaining the performance and low-level control necessary for real-time computer vision on the Jetson Nano. Although Python remains a powerful tool for testing and experimentation, our familiarity with the C++ development pipeline enables faster integration, better performance tuning, and more efficient use of system resources.

The rhythm game itself is being developed in the Godot Engine, which uses GDScript, a high-level, dynamically typed language with Python-like syntax, designed specifically for Godot's architecture. While GDScript is expressive and well-integrated with Godot's node system, it does not offer the low-level performance or external library compatibility needed for high-throughput computer vision. Therefore, GDScript will be used primarily for game logic and UI, while pose estimation will be handled externally in C/C++ and communicated to the game engine via inter-process communication (IPC) or shared memory. If deeper integration becomes necessary, Godot supports C/C++ bindings through GDExtension, enabling MediaPipe's C/C++ modules to interface directly with the engine while maintaining real-time performance.

3.9.6.1 Evaluation Criteria

The computer vision system in this rhythm-based dance game serves as an event-driven input mechanism rather than a continuously running subsystem. Its role is to detect specific player poses at designated moments defined by the game's chart or music track. While it is not active at all times, it remains an essential component during gameplay segments that involve pose-based challenges. To function effectively in this context, the system must meet key requirements in terms of latency, accuracy, robustness, and integration. These criteria ensure the pose detection system can reliably support time-sensitive interactions without introducing disruptions to the player's experience.

3.9.6.2 Latency

Timing is one of the most important aspects of rhythm game design. For players to feel connected to the beat and confident in their inputs, the delay between a physical gesture and the system's recognition of that gesture must be imperceptible. For this application, the maximum allowable end-to-end latency is defined as 100 milliseconds. This measurement begins at the moment the camera captures a frame and ends when the pose data is successfully delivered to the game engine for evaluation.

This latency budget includes image acquisition, any pre-processing applied to the frame, neural network inference for pose detection, post-processing steps like keypoint filtering or pose smoothing, and the transfer of results into the game logic. Delays beyond 100 milliseconds can create a noticeable disconnect between the player's actions and the game's response. At 150 milliseconds, players with strong rhythm sensitivity may perceive the system as being off-tempo or unresponsive.

To minimize latency, the pose detection pipeline must use lightweight models such as BlazePose or a stripped-down version of MediaPipe Pose. Frame resolutions should be kept moderate, and smoothing filters should be configured for minimal delay. Where possible, inference should be performed asynchronously on a separate thread. Hardware acceleration through CUDA or TensorRT, if compatible, can further reduce inference time and help maintain a consistent frame rate.

3.9.6.3 Accuracy

Accurate pose recognition is required to ensure the system scores player input fairly and consistently. Errors in classification, including false positives where poses are detected incorrectly or false negatives where valid poses are missed, directly impact gameplay quality. Incorrect detections can result in dropped combos, missed prompts, or undeserved penalties, reducing player satisfaction and overall trust in the game.

To address this, the pose estimation system should be configured to detect only a small set of predefined poses used in gameplay. These can be manually defined or trained using a dataset that includes common variations. Confidence thresholds should be enforced to prevent low-quality predictions from being processed. Temporal validation, where poses must be held across multiple frames to register, can help eliminate flickering caused by brief or unstable detections.

In scenarios where transfer learning is feasible, the pose detector can be fine-tuned using a small number of labeled examples that reflect the specific movement patterns and player behaviors expected in the game. This improves classification accuracy without the need to retrain an entire model from scratch.

3.9.6.4 Robustness

The game is expected to operate in a range of real-world conditions, meaning the vision system must be resilient to variability in lighting, background clutter, clothing, and camera positioning. Environments may include bedrooms, living rooms, classrooms, or arcades, each introducing different visual challenges.

Lighting changes can introduce shadows or glare. Players may wear loose, patterned, or dark clothing that affects keypoint visibility. Camera placement may vary slightly in height, distance, or tilt. To maintain stability across these conditions, the pose estimation system must be robust. Models that are invariant to scale and orientation, particularly those using heat maps or landmark regression, perform better in uncontrolled environments.

Testing should include augmenting the input video stream with artificial noise, brightness variation, and occlusion to evaluate the system's tolerance. Ideally, the pose detector should not require frequent recalibration and should support dynamic input normalization during runtime.

3.9.6.5 Integration with the Game Engine

In a rhythm-based game where player gestures are a core part of gameplay, the computer vision system must integrate cleanly with the game engine to avoid introducing delays, instability, or data inconsistency. The pose detection pipeline should operate in parallel with the main game loop, allowing the game engine to remain responsive and frame-accurate even under high processing load.

A practical approach is to run the vision system in a dedicated thread or as a separate process. This design reduces the likelihood of memory contention and helps isolate crashes or slowdowns in the vision pipeline from the rest of the game. Data such as keypoint coordinates, joint angles, or classified pose states can be communicated through shared memory buffers, local sockets, or lightweight message-passing protocols.

The interface between the vision module and the game logic should be structured and minimal. Pose data should arrive in a predictable format with consistent timing to allow the game engine to make real-time decisions, such as triggering animations, updating score counters, or initiating visual effects. Synchronization mechanisms such as frame counters, time stamps, or event queues can help ensure that pose detections are correctly matched to the corresponding frames of gameplay. When evaluating different CV pipelines, attention should be paid to their ability to support such integration models. Systems that allow for modular runtime graphs, asynchronous processing, and efficient memory management are typically better suited for integration in real-time game environments.

3.10 Game Engines

3.10.1 Godot

Godot is a very useful game engine that is beginner friendly, is open source, and has a lot of documentation as to how to use each function available in Godot. Furthermore due to Godot being open source, there are a lot of third party tutorials and walkthroughs regarding Godot teaching and guiding users how to accomplish various tasks for different types of games. These tutorials include rhythm games which is what we will be working on.

Godot's game engine seems very easy to not only install, but also use once it has already been installed. Godot's ability to create various scenes, that of which hold a variety of assets such as sprites, nodes, sound, animations and other useful assets seamlessly make it very simple to use. Godot's user interface also is very simple to understand as one does not become overwhelmed.

Godot also has its own scripting language called GDScript, which is very similar to Python which is used to be able to change various aspects of each object being made. Due to Godot's scripting language being based off of python, GDScript is a comfortable language to understand especially for game scripting beginners who do not have a lot of programming experience, specifically users without prior game design programming knowledge. Through using GDScript, there are various functionalities that can be done using it such as changing the positioning of an object or sprite across the y-axis or x-axis which we will be using for the notes that will follow the beat. Another attribute that can be changed using GDScript, are the colors of the objects after a certain button(s) is/are pressed which we will be also using for the color changes of the arrows when they are pressed.

Another fantastic feature that Godot has, is the compatibility mode that is used when creating our project. Due to us having a microcontroller that will not have as much computing power as a modern computer would have, having this as a feature will lower the load that the microcontroller would need to output. We also have a member who has used Godot in the past and due to their experience with Godot we had this option as a priority, however we checked other sources as well to test which would be best for the objective we are trying to complete.

Godot's interface is very simple to understand. The nodes that will be on a specific scene is located on the top left side where the scene tab is, the assets that are implemented throughout the entire project will be located right below that in the file system tab, and when clicking on a specific node, their properties will be located on the right side in the inspector tab.

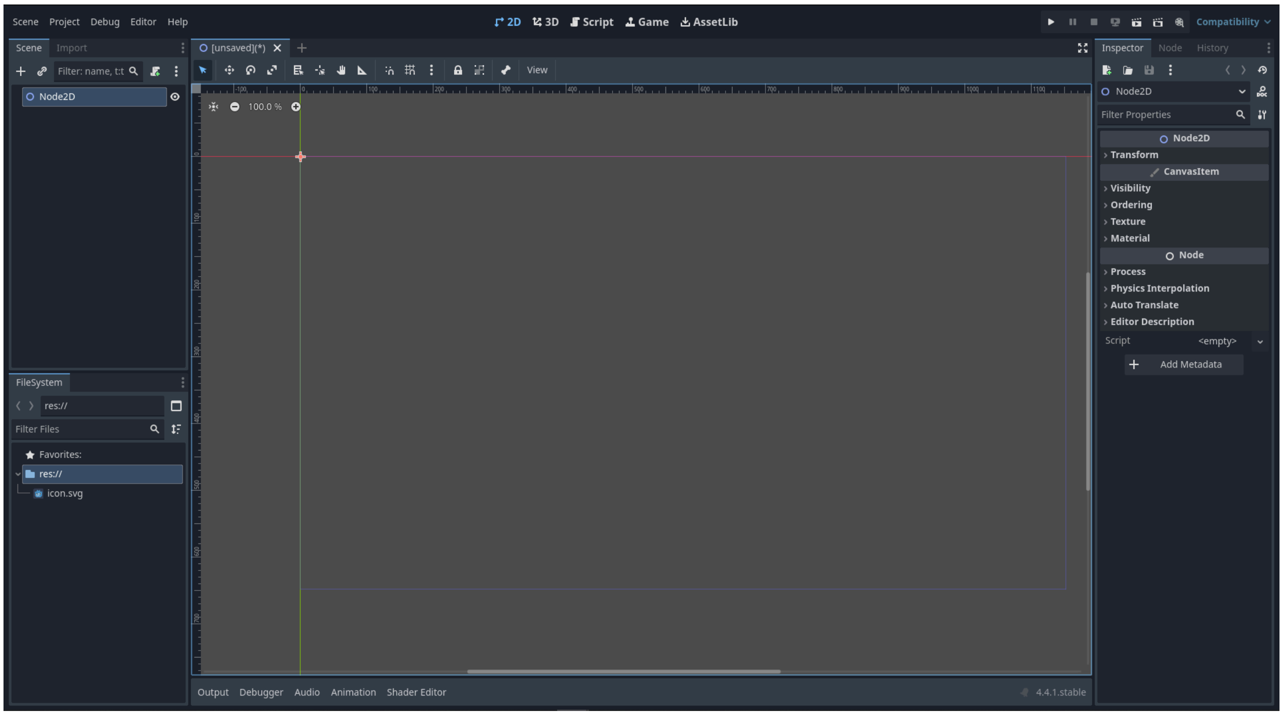


***Figure 3.11** Godot default interface*

Scene Tab

The scene tab is located by default on the top left of the Godot's user interface. The scene tab hosts all of the different nodes that are located on the specific scene that is selected. These nodes that can be added to the scene could range from different sprites, to different types of polygons some of which can add collision detection, and even audio files that can be played while the scene is being played. Afterwards, once all the different nodes are placed on the scene selected, one could add a script onto each of those individual nodes that allows modifications to the objects.

These modifications could range from the color of the polygon, the position of the sprite, or even a custom property that one chooses to add to the object. Another way that script can be attached, is by attaching them to one of the scenes as a whole. Doing so will affect any and all objects that are inside the scene. This feature makes it easier to modify a scene as a whole rather than individually modifying each object that exists in the scene and makes the process seamless.The scene tab is very useful to keep one organized as to what is going on in the scene and be able to see where everything is positioned.

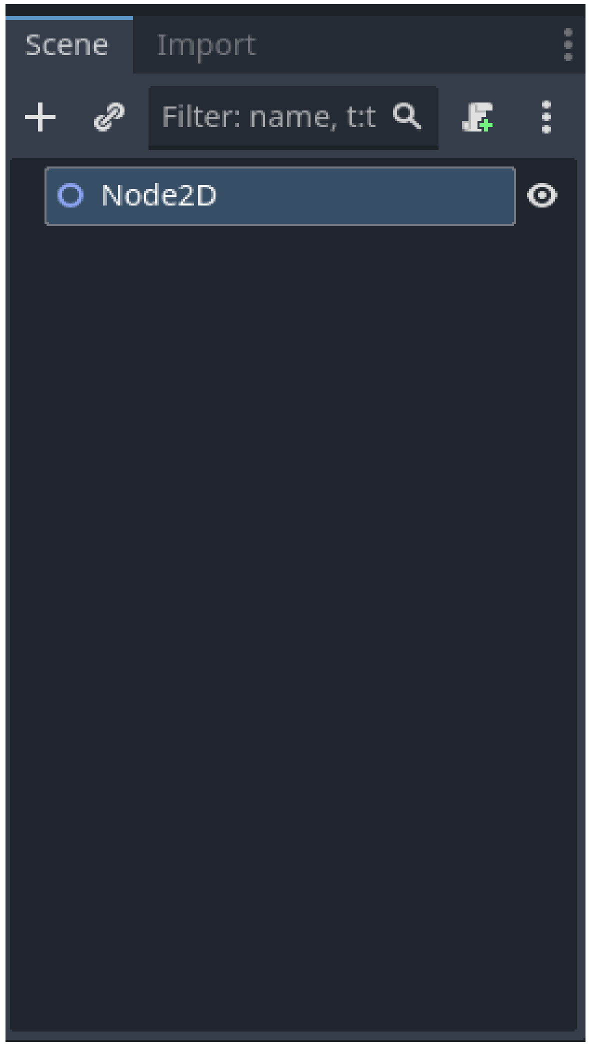


Figure 3.12 *Scene Tab*

File System Tab

The file system holds all of the assets that are needed for the project. This includes any scripts that are being run, any scenes that are made, any sprites that are imported, and much more.

The file system tab also allows for the creation of various folders for better organization for the project. Having multiple assets be bunched up makes working on it confusing, however utilizing the file system the folders make it easier to search for a specific group of assets such as one for how an arrow functions or searching for how the menu interface is set up, makes it seamless to update certain assets when needed.

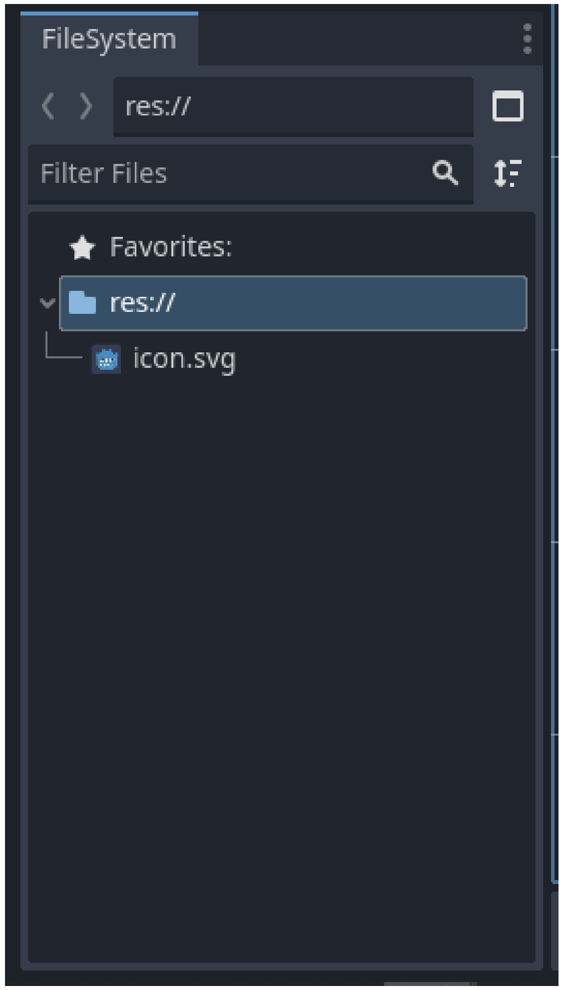


Figure 3.13 *FileSystem Tab*

Inspector Tab

The inspector tab holds all the information in regards to a certain object that is being highlighted. Whether that be an imported sprite, a custom made polygon that has the collision feature attached to it, or even an audio source that needs to play music, the inspector tab will have that information stored for users to be able to make modifications manually to certain attributes.

For example, users can decide that they would like to change the positioning of a certain sprite or polygon, under the transform dropdown one could access the various transformations that one could make such as moving across the x-axis or move across the y-axis. Users can also decide that they would like to rotate an object a certain way, this can also be done in the transform section.

There are many other attributes that users can add to the object selected through the inspector tab such as scripts that were mentioned before.

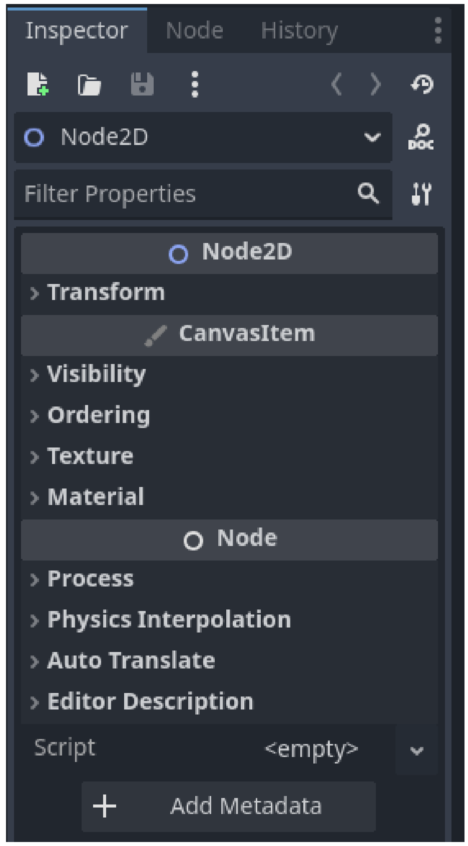


Figure 3.14 *Inspector Tab*

3.10.2 Unity

Unity is another game engine that exists that could be used to make 2D games. Unity however is more focused on 3D games rather than 2D games. Some features such as the Tilemap system, the SpriteShape, and pixel-perfect rendering exist that could help us create our 2D rhythm game that we are trying to accomplish.

Unity also offers audio synchronization tools and a timeline that could make it easier to have both the player actions along with the musical beats to be in tune, which is a key factor in terms of how the game should function while it is running.

The downside of using Unity, is that for scripting, it uses a language called C# which is a quite advanced language to use and isn't a very beginner friendly scripting language for game building. Even though it is a very powerful language to learn, its learning curve is very high without prior experience using the language.

Furthermore, due to how large unity is in terms of their features, it becomes very overwhelming understanding what unity has in store vs what it does not have in store. After opening it for the first time, Unity has a variety of libraries that come installed with it causing the engine to take a long time to open for the first time.

One good attribute that Unity has, is the built in version control called Unity Version Control. While it is a good feature to have, due to us being a group of 5 people, it would cost us an additional $14 monthly to use their built in system.

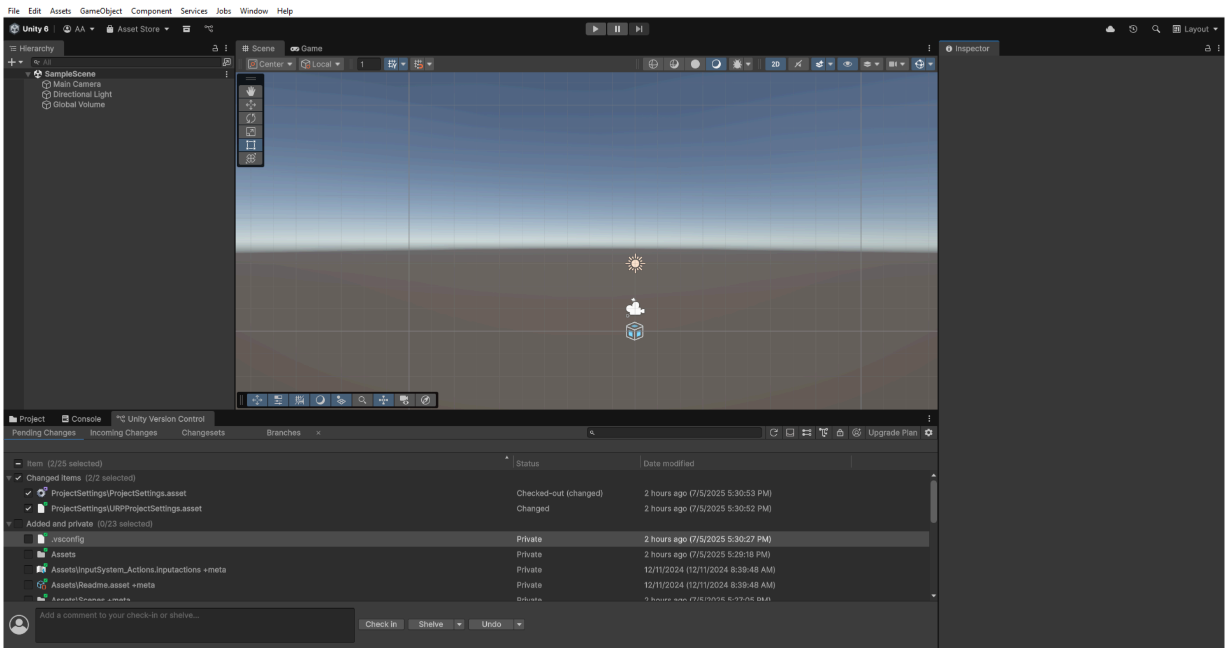


Figure 3.15 *Unity default 3D interface*

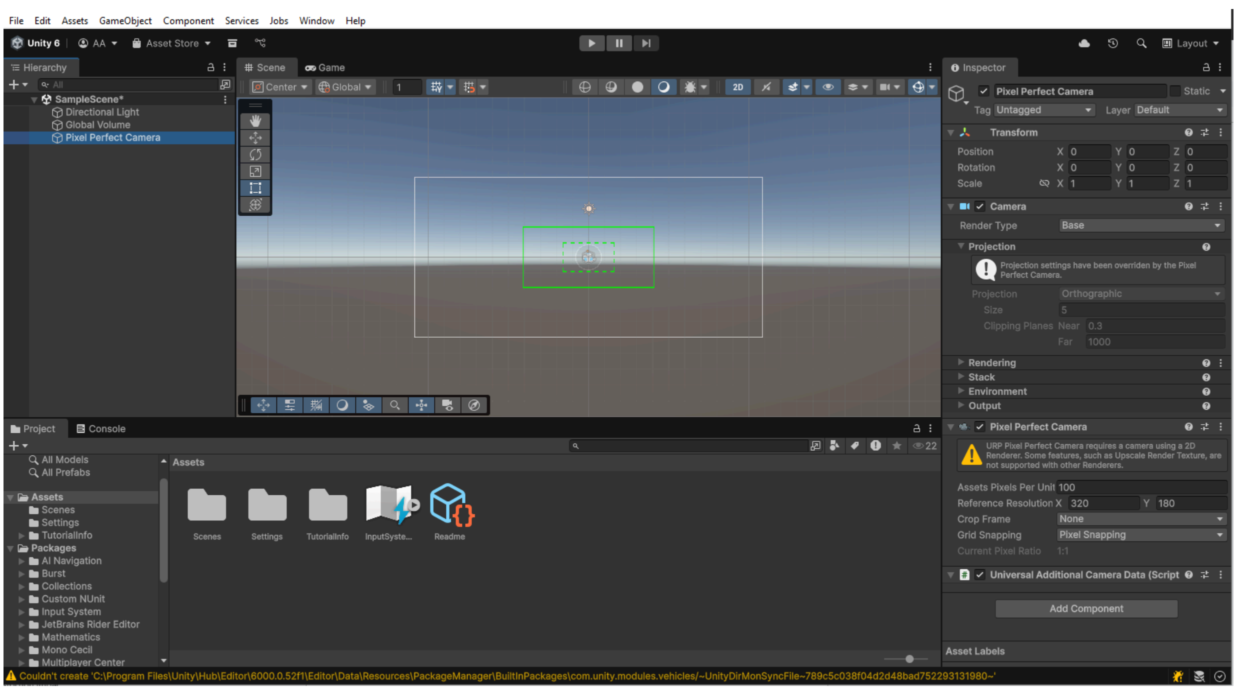


Figure 3.16 *Unity default 2D interface*

In Unity, users are able to change between the 3D version as mentioned before and a 2D version. While in the 3D version you are able to manipulate various objects needed in the X, Y, and Z directions, the 2D version only allows manipulations in a two dimensional space. The main differences between the two different versions are the objects that one can add to either version. In a two dimensional space any basic shape can be added such as circles, squares, and other 2D objects. This also includes spites and other assets that are two dimensional. In the figure there is an empty newly created project that added the two dimensional camera that is able to be moved around.

Hierarchy Tab and Inspector Tab

The Hierarchy Tab in Unity, similar to Godot's Scene Tab, is located by default on the left side of the Unity Editor interface. It serves as a structural view of all the game objects present in the current scene. These objects are organized in a parent-child relationship, allowing users to easily manage and navigate complex scene structures. From this tab, users can create new game objects, delete existing ones, and organize them to better structure gameplay logic or UI layers.

On the other hand, the Inspector Tab, found on the right side of the Unity interface, displays detailed information and configurable properties for the object currently selected in the Hierarchy. Just like in Godot, users can manipulate position, rotation, scale, and visibility, but Unity's inspector also provides deeper customization through components. Every object in Unity can be made functional by attaching components like Colliders, Rigidbodies, UI elements, AudioSources, and more.

One of the most powerful features of the Inspector Tab is the ability to attach custom C# scripts. These scripts define behaviors and can interact with other game objects and systems. Although Unity's scripting system offers flexibility and power, it comes with a steeper learning curve—especially for beginners. Understanding how different built-in methods (like Start(), Update(), or OnTriggerEnter()) interact requires time and practice. Despite this, the Inspector's modular design makes it easy to prototype features by mixing visual tools with coded logic.

Overall, the Hierarchy and Inspector tabs in Unity offer a robust combination of scene organization and detailed object manipulation, providing developers with both clarity and control throughout the development process.

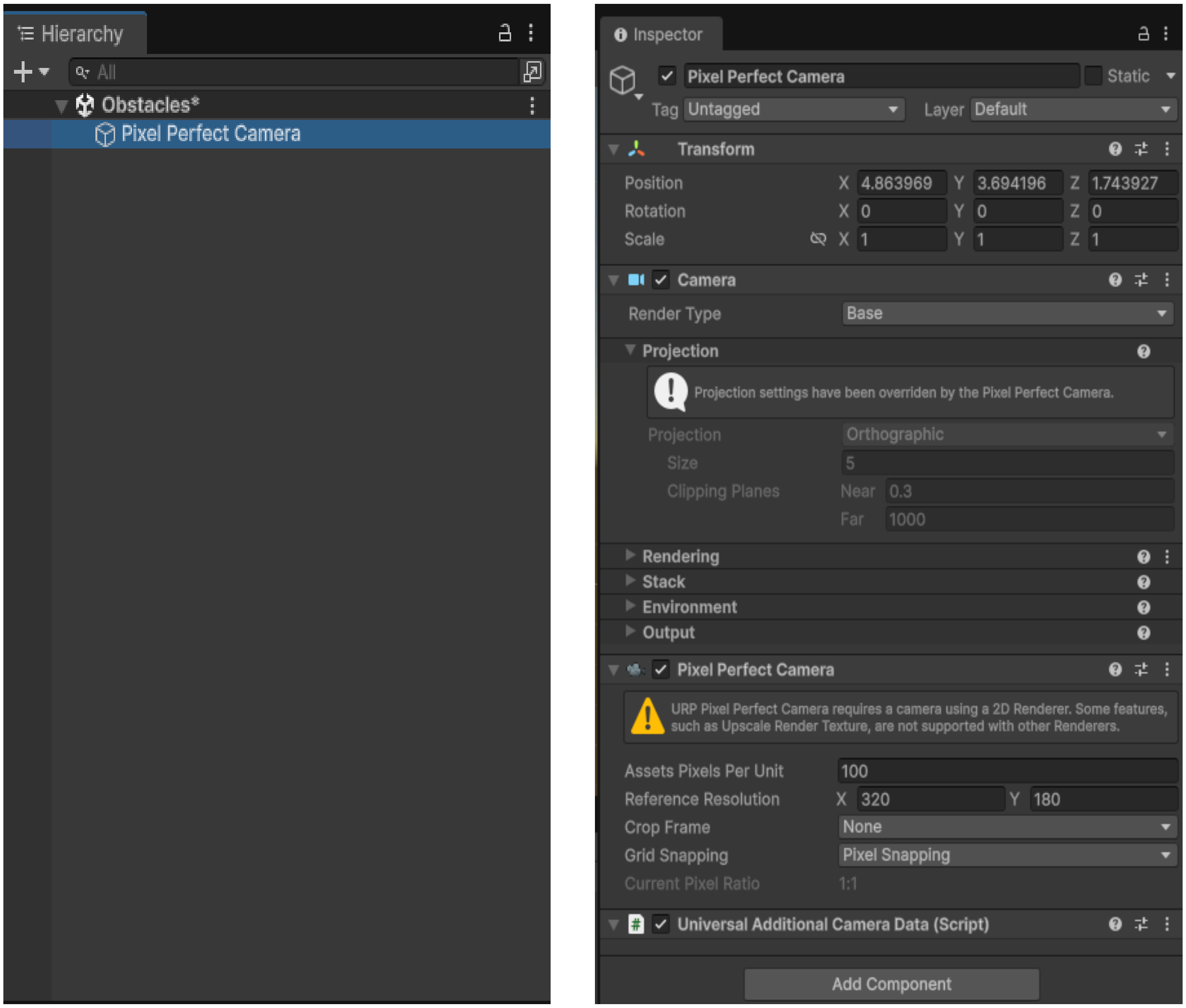


Figure 3.17 *Unity Hierarchy and Inspector Tabs*

Project Tab

Similar to Godot, Unity has the “Project Tab” where similar to Godot’s file system holds all of the assets that are needed for the project. Inside the project tab, any scripts that are being run and any assets that are imported though the Unity Asset Store can be found here. The project tab is very useful as when double clicking an object from the inspector tab, will instantly highlight and direct to you the folder from the project tab it is in.

The project tab also allows for the creation of various folders for better organization for the project. For example if there are multiple assets that have similar characteristics, they can be all sorted into a single folder, making it a great organization tool for cleaning up users workspace.

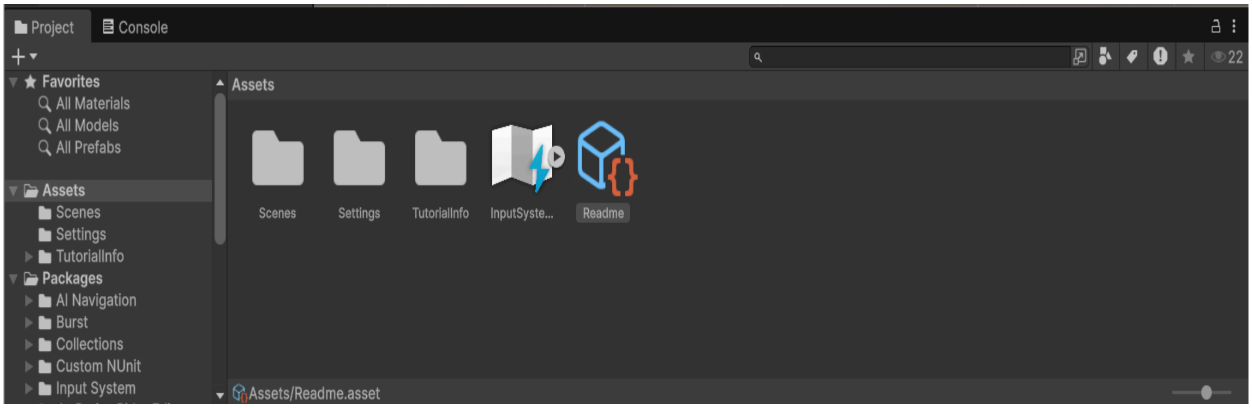


Figure 3.18 *Unity Project Tab*

3.10.3 Unreal Engine

Unreal Engine is a game engine that one can install using the Epic Games launcher created by Epic Games themselves that is mainly used for 3D games. They are mostly known for being used for AAA game development which are games that are very large with many assets attached to the game mostly made in the 3D space. However, Unreal Engine has added the capabilities to create 2D games through their “Paper 2D” functionality allowing to only see the space in the 2D perspective rather than adding the Z-direction. This functionality allows 2D sprites to be made and other 2D assets that could be used for a 2D game being developed on their game engine.

One of the downsides to using Unreal Engine however, is that although it has tutorials on how to create games, all of the ones that they offer on their website are mostly geared towards those who are aiming to create a 3D game. Since we are making a 2D game, those tutorials do not work for us entirely. There are however some aspects of those tutorials that we could use such as their scripting of sprites and event handling. Unreal Engine also has the same problem that Unity does which is the load that it has on the system.

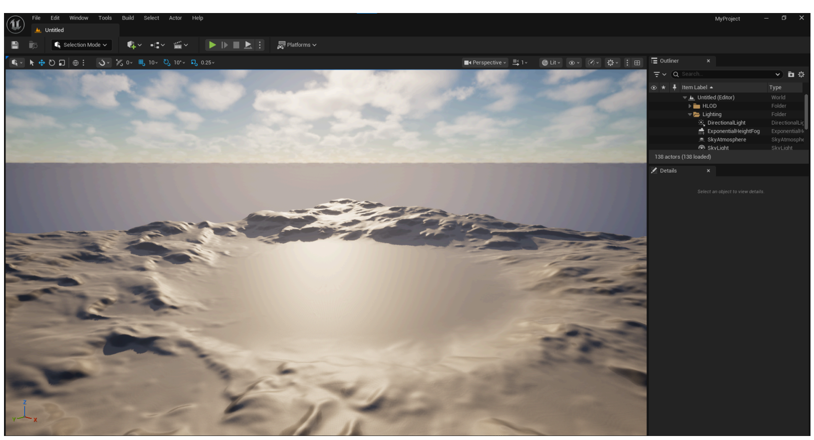


Figure 3.19 *Unity Hierarchy and Inspector Tabs*

In order to activate the 2D paper form there is a very complicated process in order for the 2D paper form to appear, however, this doesn't make the entire scene two dimensional, instead it makes the sprite themselves two dimensional and not the space that they are in themselves.

This two dimensional paper form although could be useful, since it is really only useful for three dimensional space, it would not be useful for us since we only want to work in two dimensions.

A positive aspect of using Unreal Engine is that it has its own version control feature that will be talked about later on. However although that is just one positive aspect, there are many drawbacks that would not be

Unreal although a popular option as a game engine would not be a good idea in a two dimensional space due to their lack of support for ease of use especially in the 2d space that we would need for creating our project.

3.10.4 Version Control

3.10.4.1 Github

Github is a very popular version control manager that allows multiple people to work on a project at a given time. By using their Git Repositories, users can access all of their uploaded documents through their cloud service. This allows for users to work on one computer, upload any and all documents or files they need for their project, and be able to work on another computer and retrieve all the documents they need to continue working.

Furthermore, since Github is indeed a version control manager, it is able to retrieve past versions of projects in the case that a mistake was made that caused the entire project as a whole to crash. This allows restoring the project to a previous state that it was in before the crash occurred thus recuperating the past stable needed.

However one must be careful when doing so as if not done so properly could be detrimental causing hours if not days of progress to be lost.

Another useful feature that github has is its ability to work on separate branches than the main one being used. What these branches allow one to do is work on a specific portion of the project separately using the same main project source code and make modifications without modifying the main source code. This is extremely useful for us due to the separate portions that we will be working on and not wanting to make mistakes that others would have to fix later on.

To add on to the branch, users can also merge projects together, allowing for branches that one worked on to be combined with the main project code and be updated. This allows for example one person to make changes to the main project code based on one feature, and once done, the user is able to merge with the main code that others have access to as well. One must be very careful with merging as there is a possibility of overrides that can cause projects to crash, so one must be sure that the merging procedure is done correctly so that mistakes do not appear at a later time.

Github has other resources such as its Large File Storage System and their Locking mechanism as well. Both of these resources will be integral to our project especially with having large files that if not properly maintained, could cause consequences when making future merges.

Locking allows us to be able to disallow any changes to documents unless they are unlocked, this allows for us to work on the project together without having the need to worry about accidentally making changes to the wrong file or overriding any other work that other group members have progressed on and that work being lost for good.

3.10.4.2 Perforce

When it comes to the gaming industry, the most common version control software used in the industry is actually Perforce. Perforce has a different kind of paradigm when it comes to version control with the use of its depots and streams.

Unlike git, it does not like to merge files or handle conflicts but rather focuses on having a system where only certain users are able to make changes at a time with the use of locks.
By default everything in a perforce project is read only, and when a user needs to make changes to their project they have to request to unlock it which can only be done if no one else has the key to that lock for the file.

The concept of locking for version control becomes especially useful for video game development due to the nature of binary files. Binary files are large files that humans are not able to properly differentiate changes from with examples being files such as images, videos, or 3D object files. While it may be possible to quickly see the difference from one change of file that contains code for programming and thus merge both existing changes, it would be nearly impossible to combine something like a game asset where two people ended up working on the same asset making their own unique changes.

Due to this, perforce enforces a strict policy on what files a person is able to edit at a given time, especially to prevent two people from editing a binary file.

For our project, perforce is a strong contender for collaborating on the development of our video game software. Our game will include a lot of assets such as colorful notes represented by arrows, indicators for players to do a pose, UI assets for song selection, and map files for our songs. If two of us were to step on each other's toes and accidentally both work on the same binary file, we would have to scrap either one or both of our work.

Perforce would make this scenario never happen in the first place, but it is not without its own costs. For one, perforce is a paid service and requires hosting of some sort to get going, and having more than 5 people in a project requires a hefty monthly subscription.

Additionally, Perforce is also a lot less intuitive for usual software engineers to understand and takes a lot of setup given the hosting process of it. Only one of our group members had any experience with perforce, and in the end an executive decision was made to avoid it all together due to the complicated nature of it. Instead we opted to go for a best of both worlds approach using git with file locking via a large-file-system.

While not as robust as perforce, it at the very least will not allow commits to files with locks. Furthermore, it is not a huge issue at the moment to use git since we have at most two people actively working on the development of the video game.

Perforce could be a potential addition to our project in the future if this project succeeds beyond expectations, but at the moment traditional git with large-file-system enabled will be enough.

3.10.4.3 Unity Version Control

Unity Version Control as previously mentioned is Unity’s built in version control system that allows for multiple users to access the project at any given time. With having the first 3 users free, then afterwards it will cost an extra $7 a month per additional user that one would like to add to the project.

One of the positives of using Unity Version Control is that one would not need to rely on external Version Control programs such as Git or Perforce. Having to rely on external software at times can be slightly deterring due to the fact that if the service goes down, it could cause issues on continuing to work since users would not be able to update their work in a timely manner while when using Unity Version Control, it goes directly off the users machine and can be updated at a later time without the loss of work being made.

One major drawback of using Unity Version Control is that in order to use it we are forced to use Unity as our game engine of choice. While Unity is a good game engine, there is a possibility that there are other game engines that we prefer to use.

3.10.4.4 Unreal Engine Version Control

Unreal Engine, similar to Unity, has its own built-in version control system. This is very helpful since if multiple people want to work on the project at any given time, users are indeed able to do so and are able to modify files and other assets needed as they please as long as the merging process of the multiple files are done correctly and overriding certain aspects of the project as a whole do not happen.

Like Unity, one of the positives of using Unreal Engine's built-in version control is that users of Unreal Engine would not need to rely on Git or Perforce as their choice of version control software.

One major drawback of using Unreal Engine's built-in version control is that in order to use it we are forced to use Unreal Engine as our game engine of choice. While Unreal Engine is a good game engine to use in general, for our specific project, we have not determined yet if Unreal will be our game engine of choice.

3.10.5 Game Engines Comparison

After comparing the three different game engines, Godot will be the most suitable choice for developing our 2D rhythm game. Due to its open-source nature, lightweight installation process, and their intuitive user interface, causes Godot to become especially accessible for beginners. The scripting language that Godot has built in, GDScript, closely resembles Python, which lowers the learning curve for those users who do not have prior game development experience. Additionally, Godot has extensive documentation and community-made tutorials that are extremely useful , including those specific to rhythm games, which aligns perfectly with our project's goals. Unlike Unity and Unreal Engine, which are primarily designed for 3D games and can be overwhelming with their complex interfaces and heavier system requirements, Godot offers a streamlined development environment and even includes a compatibility mode ideal for microcontroller-based projects. Given our team's familiarity with Godot and its alignment with our technical needs and project scope, we have decided to move forward using Godot as our game engine.

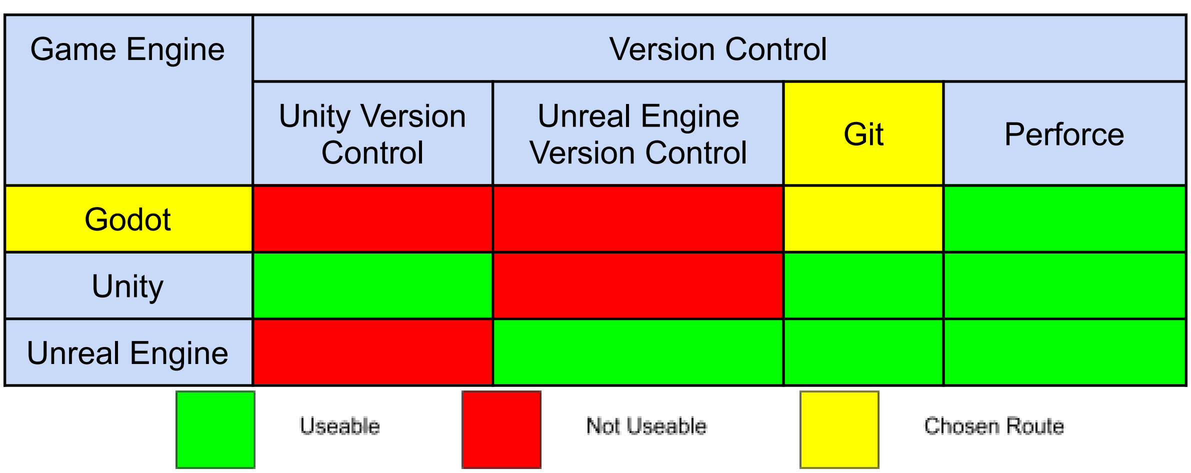
Table 3.19 *Game engines comparison table*

Feature	Game Engine		
	Godot	**Unity**	**Unreal Engine**
Primary Use Case	2D & 3D games (strong 2D support)	Primarily 3D, supports 2D	Primarily 3D, limited 2D via Paper2D
Beginner Friendliness	High	Moderate	Low

Scripting Language	GDScript	C#	C++ or Blueprints
Tutorial Availability	Extensive, especially for 2D/rhythm games	Many, mostly general or 3D-focused	Mostly 3D tutorials
Installation/Startup	Lightweight, fast and simple	Heavy install, slow to start	Very heavy, Epic Games Launcher required
UI/Workflow Complexity	Simple and clean	Complex, many default libraries	Complex and feature-dense
Audio/Timing Features	Manual via scripting	Built-in audio sync and timeline tools	Available but requires additional setup
Version Control Support	External tools (e.g., Git)	Built-in (costs extra for teams)	External tools (e.g., Git)
Open Source	Yes	No	No
Microcontroller Friendly	Yes, has compatibility mode	No	No
Best Use Case in Project	Ideal for 2D rhythm game with limited resources	Possible, but better for Unity experts	Least suitable for 2D rhythm game beginners

3.10.6 Version Control Comparison

Table 3.20 *Version Control Comparison Table*



After comparing the version control options that we had available, since we have decided to use Godot as our game engine of choice, we had the option to choose between Perforce and Git. Ultimately we have decided to use Git as our version control of choice due to its various features such as locking, large file storage system or LFS for short, as well as its ability to have multiple users work on the project and be able to upload their work by merging with existing files.

As a result we have Godot and Git as our core game engine and version control systems to complete our goal with the dance rhythm game system we plan to create.

4. Standards and Design Constraints

Standard: IPC-2221A – Generic Standard on Printed Board Design

The IPC-2221A, "Generic Standard on Printed Board Design," serves as the foundational standard for the design of printed circuit boards. Published by IPC, the global electronics industry association, this document establishes the generic requirements for the design of organic printed boards, from single-sided boards to complex multilayer structures. It provides a robust framework of rules and principles that govern nearly every aspect of PCB layout, ensuring that the final product is not only functional and reliable but also manufacturable. For any electronic design project, applying the principles of IPC-2221A is a critical step in translating a schematic diagram into a physically sound and dependable product.

At its core, IPC-2221A is a guide for Design for Manufacturability. It creates a common language and a set of baseline requirements between the designer and the fabrication house. By establishing rules for spacing, feature sizes, and tolerances, the standard ensures that a design can be reliably produced using standard manufacturing processes, which helps to control costs, improve yields, and reduce production time. The standard also defines three performance classes based on the intended end-use environment, which dictates the stringency of the design rules. Class 1 is for general electronic products where the primary requirement is the function of the completed assembly. Class 2 is for dedicated service electronic products requiring high reliability and an extended service life, where uninterrupted service is desired but not critical. This is the target class for the dance pad project. Class 3 is for mission-critical products where continued high performance or performance-on-demand is essential.

IPC-2221A provides extensive guidance on the electrical aspects of PCB design to ensure both signal integrity and user safety. One of the most critical safety considerations in PCB design is the spacing between conductive elements, known as clearance. Insufficient spacing can lead to dielectric breakdown or arcing between traces, especially at higher voltages. IPC-2221A provides detailed tables that specify the minimum required clearance based on the peak DC or AC voltage between conductors. These requirements vary based on whether the conductors are on internal or external layers and whether they are coated. For the dance pad, which operates at a low voltage of 5V, the minimum spacing requirements are easily met, but acknowledging this standard is crucial for demonstrating sound design practice.

The width and thickness of a PCB trace determine its current-carrying capacity. A trace that is too small for the current it must carry will overheat due to its own resistance, which can damage the PCB laminate or cause the trace to fail entirely. IPC-2221A provides charts and formulas that relate a conductor's cross-sectional area to its temperature rise for a given amount of current. This allows designers to select an appropriate trace width to ensure the board operates within safe thermal limits. For the power and ground traces on the dance pad's PCB, these guidelines were used to ensure they could handle the total current draw of the system without significant heating.

Beyond electrical rules, IPC-2221A specifies the physical and mechanical characteristics of the board to ensure its structural integrity and compatibility with assembly processes. Vias are plated-through holes that form electrical connections between different layers of a PCB, and their reliability is paramount to the function of a multilayer board. IPC-2221A provides specific guidelines for via design. The annular ring is the ring of copper that surrounds a drilled hole. The standard specifies a minimum acceptable annular ring width to ensure a solid connection between the via barrel and the trace after drilling and plating, accounting for manufacturing tolerances. For Class 2 designs, a robust annular ring is required to prevent breakout, where the drill hole is not completely surrounded by the copper pad. The aspect ratio is the ratio of the board's thickness to the diameter of the drilled hole. A high aspect ratio can make it difficult to achieve reliable copper plating down the entire barrel of the via, so the standard provides limits on aspect ratios to ensure manufacturability.

The standard also addresses the need to manage heat generated by components. A common technique guided by IPC-2221A is the use of thermal relief pads. When a component pin needs to connect to a large copper plane, such as a ground plane, a direct connection would act as a large heat sink, making it very difficult to solder the component. A thermal relief pad creates small copper spokes to make the connection, which reduces the heat transfer during soldering while still providing an adequate electrical connection.

A critical function of the IPC-2221A standard is to define the requirements for a complete and unambiguous documentation package. A design is only as good as the documentation that communicates it to the manufacturer. The standard calls for a set of documents that typically includes a fabrication drawing, an assembly drawing, Gerber files, and a Bill of Materials. The fabrication drawing specifies the board's dimensions, layer stack-up, materials, and drill hole information. The assembly drawing shows the location and orientation of all components. Gerber files are the industry-standard file format that describes each layer of the PCB, such as copper layers, solder mask, and silkscreen. The Bill of Materials is a complete list of all components to be mounted on the board. By following the documentation guidelines of IPC-2221A, the design intent for the dance pad's electronics is clearly and professionally conveyed, minimizing the risk of manufacturing errors.[30][31][32]

Standard: Universal Serial Bus (USB)

The Universal Serial Bus is a ubiquitous industry standard that establishes specifications for cables, connectors, and communication protocols for connection, communication, and power supply between computers and peripheral devices. Its development was intended to standardize the connection of computer peripherals, thereby replacing a multitude of legacy interfaces. For this project, adherence to the USB standard is critical for ensuring interoperability and a seamless user experience.

While several revisions of the USB standard exist, the USB 2.0 specification was selected for this project due to its widespread compatibility and sufficient data throughput for the required application. USB 2.0 provides a maximum data signaling rate of 480 Mbit/s, which is more than adequate for transmitting the state changes of the dance pad's nine input tiles. Furthermore, the standard specifies a bus power supply of 5V, with a high-power device permitted to draw a maximum of 500mA, a critical constraint for the system's power budget.

A key component of the USB specification relevant to this project is the Human Interface Device class. The HID class is a device class specification that defines a protocol for low-latency, low-power peripherals, such as keyboards, mice, and game controllers. The primary advantage of utilizing the HID class is that it obviates the need for custom host-side drivers, as support is natively integrated into modern operating systems, including Windows, macOS, and Linux.

Device functionality is communicated to the host computer through a series of descriptors during the enumeration process. Device descriptors identify the peripheral with a unique Vendor ID and Product ID. HID descriptors specify that the device conforms to the HID class. Report descriptors are the most critical for this project, as they define the structure and format of the data packets, known as reports, that the device will send to the host. For the dance pad, the report descriptor will be configured to define a data structure containing the boolean state of each of the nine input tiles, effectively representing them as buttons. Once enumerated, the device transmits data to the host via Input Reports over a dedicated interrupt-based endpoint, ensuring that state changes are communicated with minimal latency.[33][34]

Standard: FCC Part 15 – Unintentional Radiators

The Federal Communications Commission establishes regulations to manage electromagnetic interference under Title 47, Part 15 of the Code of Federal Regulations. This standard governs the operation of radio frequency devices without requiring an individual license. Any electronic device incorporating digital logic and operating with clock frequencies greater than 9 kHz is capable of generating electromagnetic energy. While not its primary purpose, this energy can radiate from the device and cause interference with radio communications. Such devices are classified by the FCC as unintentional radiators.

FCC Part 15 defines two primary classifications for unintentional radiators based on the intended market and environment. Class A is for devices used in commercial, industrial, or business environments, where the limits on radiated and conducted emissions are less restrictive. Class B is for devices intended for use in residential environments. The emissions limits for Class B are significantly more stringent to prevent interference with consumer electronics such as televisions, radios, and Wi-Fi networks. Given that the 9-tile dance pad is a consumer-oriented gaming peripheral, it falls under the Class B classification. Therefore, the design must consider stricter emission limits applicable to residential devices.

While formal FCC certification is beyond the scope of this academic project, incorporating design principles aimed at minimizing electromagnetic interference is a fundamental aspect of professional engineering practice. Several strategies have been considered in the design of the device's printed circuit board and overall system architecture. The PCB is designed with a large, contiguous ground plane, which provides a low impedance return path for digital signals and minimizes the area of current loops that can act as radiating antennas. Small ceramic decoupling capacitors are placed physically close to the power and ground pins of the microcontroller and other integrated circuits. These capacitors serve as a local charge reservoir and shunt high-frequency noise from the power distribution network to the ground plane, preventing its propagation. The slew rates of high-speed digital signals are controlled where possible, as sharper signal transitions contain higher-frequency harmonic content that can contribute to radiated emissions. Finally, a shielded USB cable will be specified for connecting the device to the host computer. The shield helps to contain common-mode noise generated by the device's circuitry and prevents the cable from acting as an antenna. By implementing these EMI mitigation techniques, the design proactively addresses the requirements of FCC Part 15, ensuring the device operates as a responsible and non-interfering electronic product.[35][36]

Standard: UL 62368-1 – Hazard-Based Safety for ICT & AV Equipment

The UL 62368-1 standard represents a modern, hazard-based approach to product safety for Information and Communication Technology and Audio/Video equipment. Published by Underwriters Laboratories, this standard shifts from a traditional incident-based model to a proactive, performance-based methodology focused on identifying potential hazards and implementing safeguards to mitigate them. As the dance pad is an electronic peripheral that directly interfaces with both a host computer and a human user, ensuring its safety in accordance with established principles like those in UL 62368-1 is a primary design objective.

UL 62368-1 is built on the principles of Hazard-Based Safety Engineering, which involves a three-step process: pinpointing all potential energy sources within the product, categorizing the energy sources based on their potential to cause pain or injury, and applying appropriate safeguards to protect users from any energy sources classified as hazardous. For the dance pad project, the primary energy sources are electrical and mechanical.

While this academic project will not undergo formal UL certification, the design philosophy of UL 62368-1 has been integrated into the development process to ensure user safety. Electrical hazard mitigation is addressed by powering the device exclusively by a 5V DC source via a standard USB 2.0 port, which is classified as an ES1 environment under UL 62368-1. This low voltage is not considered hazardous and does not present a risk of electric shock. To protect against potential short circuits or overcurrent conditions, the custom printed circuit board design incorporates a resettable Polymeric Positive Temperature Coefficient fuse. This device automatically interrupts the circuit in an overcurrent event and resets when the fault is cleared, providing robust protection against fire hazards. All internal wiring is appropriately insulated and secured to prevent abrasion or pinching.

Mechanical hazard mitigation is also considered, as the dance pad is subjected to significant and repeated mechanical stress during gameplay. The enclosure is constructed from high-impact polycarbonate and plywood, materials selected for durability and resistance to fracture. All external corners and edges of the enclosure are rounded to a radius of no less than 3mm to prevent cuts or injuries from sharp edges. The internal structure is designed to distribute impact forces, ensuring the assembly remains mechanically sound and that no internal components can become dislodged and create a secondary hazard. By adopting the hazard-based principles of UL 62368-1, the design of the dance pad prioritizes user safety, addressing potential electrical and mechanical risks through deliberate material selection and the implementation of appropriate safeguards.[37][38]

Standard: RoHS – Restriction of Hazardous Substances

The Restriction of Hazardous Substances Directive, originating in the European Union, is a critical environmental and health-focused standard for the electronics industry. The directive, specifically RoHS 3, restricts the use of ten specific hazardous materials in the manufacture of various types of electrical and electronic equipment. Although this project is not intended for commercial sale in the EU, voluntary adherence to RoHS principles represents responsible engineering practice, minimizing environmental impact and ensuring the final product is safe for users and for disposal.

The RoHS directive restricts several substances to a maximum concentration of 0.1% by weight, apart from Cadmium, which is limited to 0.01%. These substances include Lead, Mercury, Cadmium, Hexavalent Chromium, Polybrominated Biphenyls, Polybrominated Diphenyl Ethers, and four specific Phthalates.

A conscious effort was made throughout the procurement and assembly phases to ensure the components and materials used in the dance pad are RoHS compliant. All active and passive electronic components, including the microcontroller, resistors, capacitors, connectors, and diodes, were sourced from reputable distributors where they were explicitly verified to be "RoHS Compliant" based on manufacturer's datasheets. This ensures that the fundamental building blocks of electronics are free from restricted hazardous substances. The custom PCBs for the project were manufactured by a fabrication house that offers a RoHS-compliant manufacturing process, guaranteeing that the PCB substrate, solder mask, and surface finish do not contain restricted materials. The most significant step taken to ensure RoHS compliance during in-house assembly was the exclusive use of lead-free solder. Instead of traditional tin-lead solder, a tin-silver-copper alloy was used for all soldering tasks. While lead-free solder requires higher working temperatures and presents different wetting characteristics, its use eliminates the most common hazardous substance found in electronics assembly, protecting both the assembler and the environment. By adhering to the RoHS directive, this project demonstrates a commitment to modern, environmentally conscious design standards that extend beyond immediate functional requirements.[39][40]

Standard: IPC-A-610 – Acceptability of Electronic Assemblies

The IPC-A-610, "Acceptability of Electronic Assemblies," is the most widely recognized global standard for the workmanship and quality of printed circuit board assemblies. Developed by the Association Connecting Electronics Industries, this standard provides comprehensive, visually supported criteria for evaluating the quality of electronic assemblies. Adhering to the principles outlined in IPC-A-610 is essential for ensuring the long-term reliability and functionality of the dance pad custom electronics. A failure in a solder joint or a misplaced component could lead to intermittent or total failure of the device during use.

IPC-A-610 defines three classes of product quality, reflecting the intended life cycle and operational environment of the assembly. Class 1 is for general electronic products where the primary requirement is the function of the completed assembly. Class 2 is for dedicated service electronic products requiring extended reliability and continued performance, where uninterrupted service is desired but not critical. Class 3 is for high performance or harsh environment electronic products where continued high performance or performance-on-demand is critical. For this project, Class 2 was established as the target quality standard. As a game controller subject to physical impact and repeated use, a high degree of reliability is necessary to ensure a consistent user experience.

The criteria of IPC-A-610 were applied during the hand-assembly and inspection phases of the custom electronics. All solder joints were created with the goal of meeting Class 2 acceptability criteria. This includes ensuring proper wetting on both the component lead and the PCB pad with a contact angle of less than 90 degrees, forming a concave solder fillet, and avoiding common defects such as cold joints, disturbed joints, excess solder, or solder bridging between adjacent pads. Meticulous care was taken during component placement to ensure correct orientation for polarized components, such as diodes and electrolytic capacitors. All components were placed centrally on their respective land patterns on the PCB before soldering, as specified by the standard. Following assembly, each board underwent a thorough visual inspection using magnification and proper lighting. This inspection verified that solder joints were acceptable and that no foreign object debris, such as stray wire clippings or solder balls, was present on the board. Furthermore, the boards were cleaned with isopropyl alcohol to remove flux residue, which can be corrosive over time and can lead to electrical leakage paths, in alignment with IPC standards for post-assembly cleanliness. By targeting IPC-A-610 Class 2 workmanship, the project ensures that the assembled electronics are not only functional but also robust and reliable, capable of withstanding the demands of their intended application.[41][42]

4.1 IEC 62471 - Photobiological Safety

When designing the LED panels, it was important to verify whether the system adhered to relevent photobiological safety standards. IEC 62471 is an internationally recognized framework standard that defines safety evaluation criteria for optical radiation, including visible, ultraviolet, and infrared sources. The purpose of this standard is to ensure systems protect human eyes and skin from photobiological hazards associated with artificial light sources, such as LEDs. IEC 62471 classifies systems into risk groups based on their radiometric output and potential health hazards, ranging from Exempt (RG0) to Risk Group 3 (high risk). [14].

For S.T.E.P.S, 850 nm infrared LED strips are being used to provide active illumination for pose detection. Since 850nm is near-infrared and partially outside of the visible range, it is still covered by IEC 62471, which encompasses wavelengths between 200 nm and 3000 nm.[14] The LEDs being used are placed approximately 1.8 meters from players. According to industry summaries of IEC 62471, systems using 850nm LEDs typically fall within Risk Group 1 provided their radiant exposure stays below the established safety thresholds. [14]

Although minimizing the radiant flux is recommended to comply with IEC 62471, in this project the actual risk is very minimal. The 850nm LEDs are partially visible, mostly safe, and extremely common in consumer motion-tracking systems. The player distance of 1.8meters, combined with time-multiplexing and wide distribution of LEDs, means that the design itself is very low risk and well within Risk Group 1. No hazardous exposure conditions are expected.

4.2 ISO 9241-210 - Ergonomics of human-system interaction

Another important standard was ISO 9241-210. ISO 9241-210 defines principles for designing interactive systems with a focus on user comfort, safety, and effectiveness.[13] This standard is relevant to the illumination system because it emphasizes minimizing discomfort. Glare, and distractions. It is important that the system does not affect the player’s comfort. Avoiding glare and minimizing distracting illumination effects is important for user comfort and effectiveness. The system will follow ISO 9241-210 by ensuring the LEDs do not produce visually uncomfortable brightness (by selecting 850nm over visible white), by time-multiplexing zones to prevent perceptible filcker, and by mounting the arrays to avoid direct illumination into the players’ eyes. Together, these measures support a user-centered design that prioritizes both an enjoyable and non-distracting gameplay experience.

4.3 IEC 60598 Luminaires

The final standard considered is IEC 60598. IEC 60598 specifies safety requirements for luminaires with respect to electrical, thermal, and mechanical hazards[15]. Although originally written for traditional lighting fixtures, its principles remain relevant due to the LED arrays function as a luminaire in the player environment. The system will follow these guidelines by ensuring properly rated wiring, insulation, and current-limiting fuses to prevent electrical shock, overheating, or mechanical failure. Given that the proposed 12V LED illumination operates at low voltage with moderate current, no additional IEC 60598 certification is anticipated beyond following standard best practices.

Finally basic electromagnetic compatibility considerations (EMC) practices were considered to ensure the LED driver circuitry does not interfere with the camera system. The PCB will include standard layout practices such as decoupling capacitors and solid grounding to reduce any risk of electrical noise affecting camera performance.

Table 4.1 *Optical and illumination standards*

Standard	Purpose	Relevant Wavelength	Risk Level In S.T.E.P	Design Impact
IEC 62471	Photobiological LED Safety	200-3000nm	RG1 (Low Risk)	850nm IR LED selection, distance testing
ISO 9241-210	Human-system interaction ergonomics	Visible spectrum	N/A	Zone lighting layout, glare avoidance
IEC 60598	Electrical safety for luminaries	Electrical/thermal	N/A	PCB layout, fuse integration

4.4 Optical Design Constraints

The main design constraint for the optical system is achieving a sufficient horizontal field of view (HFOV) and depth of field (DOF) to cover the entire dance pad array. The system must monitor a 2.9 m * 2.9 m play area from a camera mounted approximately 1.8m away. This geometric requirement translates to a horizontal field of view of approximately 93°, based on trigonometric calculations. Maintaining this coverage ensures that all nine dance pads remain consistently visible to the pose estimation algorithm at all times. Minimizing the possibility of tracking errors or missing key player landmarks. To achieve this constraint, a 3mm focal length aspherical lens was designed, balancing the need for wide-angle capture with minimal distortion and acceptable pixel resolution. This design choice guarantees that the system can consistently and reliably observe the entire interactive area without gaps, which is essential for gameplay accuracy and user satisfaction.

With a 93° FOV covering the dance pad array, each pad will occupy a relatively small number of pixels in a 1080p frame. This means there will be a trade-off between covering a large area and maintaining enough pixel density for precise pose detection. The 3 mm aspherical lens balances this trade-off by providing wide coverage while maintaining acceptable pixel density to allow MediaPipe to resolve key landmarks with adequate clarity. This constraint ensures that gameplay remains responsive and reliable, without introducing latency or missed detections.

The physical placement of the camera module is another important constraint. The camera must be positioned to achieve the target field of view while avoiding occlusions caused by player movements, and minimizing distortion. The camera is planned to be mounted at the average torso height of approximately 1-1.5m above the floor, and angled downward to encompass the entire 2.9 m * 2.9 m player tracking area, This position balances both the horizontal and vertical FOV requirements while maintaining a natural perspective of the player's posture. It also helps minimize motion blur and ensures that MediaPipe receives high-quality data without key body joints being blocked by sudden movements. Consistent, predictable mounting geometry is essential to calibrate the tracking system and maintain performance between game sessions.

Uniformity of illumination is a critical design constraint to ensure that MediaPipe's AI pose estimation functions accurately across the entire play area. A non-uniform lighting pattern with hotspots or dark corners could cause the system to lose track of the player, reducing gameplay accuracy. An issue that could arise in dimly lit rooms such as some arcade enivronments. The target uniformity is specified as at least 85%, ensuring that the lowest illuminated areas maintain at least 85% of the brightness of the most illuminated regions. To achieve this, the LED layout was designed to distribute light evenly, with time-multiplexed zone activation reducing local overheating and maintaining consistent brightness throughout the tracking space. During testing, uniformity will be verified by analyzing pixel intensities of a calibration target using the camera sensor, as a practical substitute for more complex photodiode measurements. This constraint supports a robust, repeatable, and comfortable user experience.

The power consumption of the LED-based illumination system was considered to ensure the system operated within reasonable current limits while maintaining efficient performance. Time-multiplexing was adopted to keep the instantaneous current per zone modest, simplifying the PCB layout and reducing any potential thermal buildup. By activating only one lighting zone per camera frame (at 120 fps with an 8.3 ms frame duration), and cycling through all four zones every 33 ms, the system limits instantaneous current draw 1.5A per zone instead of a peak 6A if operated simultaneously. This strategy supports simpler, more reliable driver electronics while ensuring consistent illumination without perceptible flicker for the pose detection algorithm.

In addition to technical constraints, practical considerations such as development time, budget limitations, and component availability also influenced the optical system design. Considering the project operates within a constrained academic timeline of two semesters, the number of design iterations is limited. Creating a fully functioning prototype within that time frame emphasizes the need for early, informed component selection. Economically, the system had to remain affordable, which excluded expensive custom optics or high-end machine vision cameras. Off-the shelf M12 lenses and commercially available LED modules were selected to strike a balance between cost and performance. Furthermore, all components had to be available lab tools and fabrication resources,minimizing custom machining or advanced calibration procedures. These constraints shaped the design process, ensuring that the final optical system remains technically sound and feasible to replicate with a modest engineering budget and time frame.

Table 4.2 *Optical and Illumination Design constraints*

Design Constraint	Requirement	Justification/Notes
Horizontal Field of View (HFOV)	≥ 93° to cover 2.9 m x 2.9 m player tracking area at 1.8 m distance	Ensure entire play area is visible without blind spots or occlusion
Focal Length	~ 3 mm (aspherical M12 lens)	Balances wide coverage with acceptable pixel resolution (≥3pixels/mm)
Pixel Density	≥ 3 pixels/mm	Required to resolve ~1mm player features for MediaPipe tracking
Camera Height & Angle	1.0-1.5m, angled 15-20° downward	Captures full body while minimizing occlusion and distortion
Illumination Uniformity	≥ 85% brightness uniformity across play area	Prevents tracking dropout due to hotspots or dark zones
Power Consumption	≤ 1.5 A per LED zone, ≤ 18 W per zone	Ensures thermal safety and simplifies driver circuitry with time-multiplexing

Design Constraint	Requirement	Justification/Notes
Time-Multiplexed IR Zones	One zone active per frame (60 FPS → 16.7 ms per frame, 4-zone cycle = 66.8 ms)	Avoids flicker and reduces peak current draw while maintaining seamless illumination
Budget	Off-the- shelf components only	Avoids costly custom optics and enables faster, more reliable prototyping
Development Time	Completed within 2-semester academic schedule	Limits iteration cycles, emphasizing early, well-informed design decisions.
Component Availability	Compatible with available lab equipment and tools	Reduces need for machining or specialty alignment fixtures.

4.5 Design Trade-off: Full Coverage vs. Low Distortion

4.5.1 FOV Lens comparison

A critical decision in the optical system design was selecting between two M12 lenses: the CIL034 (3.2 mm, 87° HFOV) and the CIL329 (2.8 mm, 120° HFOV). From the fixed camera mounting distance of 1.83 m , the system requires a horizontal field of view of at least 93° to fully encompass the 2.9 m- wide tracking area. While the 120° lens easily satisfies this requirement, it introduces significant barrel distortion near the edges of the frame. This distortion can negatively impact pose estimation by warping limb proportions or causing landmark misinterpretation near the periphery.

In contrast, the 87° lens provides lower optical distortion and higher pixel density across the player's body, enhancing the quality of MediaPipe's pose recognition, particularly in the central tracking zone. Although it offers slightly less edge margin than wider-angle options, it still captures the full dance pad area from the 1.8m distance with minimal distortion..

After evaluating these trade-offs, the 3.2 mm (87°) lens was selected due to its sharper central imaging, lower distortion profile, and sufficient coverage for solo gameplay, where players tend to remain within the central region. This design choice prioritizes pose detection fidelity and image quality over more peripheral buffer coverage, a compromise deemed acceptable for the current gameplay design. Future revisions may consider distortion-tolerant tracking models or higher resolution sensors to accommodate wider FOVs without compromising detection accuracy.

4.5.2 Distortion Impact on AI Performance

The selected CIL034 lens specifies <0.1% distortion and is classified as rectilinear, meaning it maintains straight lines across the field. This helps reduce issues such as wrapping of limbs or landmark displacement at the edges of the frame. Although distortion is more pronounced in ultra-wide lenses like the 120° model, the chosen 87° lens ensures spatial consistency in landmark tracking. While MediaPipe has some tolerance for distortion, retaining geometric fidelity in the player's body structure improves joint prediction confidence and minimizes tracking errors. If wider coverage is required in future iterations, software correction (e.g. OpenCv lens calibration) could be considered to compensate for edge warping without hardware changes.

5. Comparison of AI Engines

5.1 Case Studies

The goal of using various Artificial Intelligence engines is to see various aspects of information given by Artificial Intelligence and compare them with one another. One main question that was asked to the 4 engines used was: What would be the best way to create a rhythm game for a beginner? Would using a game engine be best or are there other alternatives to complete this objective? The reason for the prompt was to find out what would be the best way to create a rhythm game for a project of this magnitude. We have heard of game engines that existed just as Godot, Unity, Unreal Engine, and Scratch, however we wanted to see which of those listed are beginner friendly. Furthermore we wanted to see if Artificial Intelligence would have recommended any other sources that aren't game engines that exist that are also beginner friendly that could be used to accomplish our objective. What we found were the following responses from each Artificial Intelligence engine when given the prompt above.

Some more case studies we can look at are regarding our hardware. While we know that we need an MCU, we want to see what MCU works best for a design, so we also asked the 4 engines the following: What MCU would be best to run a 9 directional dance pad that can connect to a PC via USB, while also being able to communicate to an LED driver to switch an LED panel on and off? The purpose of this prompt is to see what our best options are and how they are the best options, in order to narrow down our research as there are multiple options to choose from.

Another case study we can look at is regarding our dance pad panel sensors. Much like our MCU, we asked the 4 engines the following: What type of dance pad panel sensor is best used in arcade grade dance pads and DIY dance pads? The purpose of this prompt is to see what is typically used so that we have a general idea of what the most common types are, so that we can narrow down our research.

The other case study that we asked each of the different Artificial Intelligence large language models was: How can we implement computer vision into a dance rhythm game using a camera? The reason for this was to choose how we can implement existing computer vision technology into the game after.

The last case study we asked to the Large Language Models was: How can we implement rhythm synchronization into a Godot project?

5.2 ChatGPT

After looking at ChatGPT's response to the rhythm game question asked, some very helpful information was given in regards to what we can use for designing the game. ChatGPT said that game engines are a recommended resource that can be used for those who are coding games as a beginner. Game engines are able to help users with difficult portions of game design with aspects such as inputs and audio being hard to use . The ones that ChatGPT recommended were Unity and Godot with scratch being added as well. [23]

Regarding MCU selection, ChatGPT offered a list of MCU with their pros and cons, as well as what they are best for. This is a good thing because it helped narrow our research better. It also goes over LED Driver Options, and USB HID vs Serial, little things to keep in mind. Lastly, ChatGPT recommended Teensy 4.0 and Arduino Leonardo for plug-and-play USB controller functionality, responsive LED control, and easy development. However, Raspberry Pi Pico or STM32F103 (Blue Pill) is best if we want more power and flexibility. [23]

For our dance pad sensor question, we got a list of the most commonly used and presumably most effective panel sensor types. ChatGPT listed each type, what they're best for, where they are used in, how they work, and pros and cons. ChatGPT does pretty well summing up the key features of each sensor, as well as providing useful information. However, the provided information is still general, mainly used as a base for research. After all the listed types of sensors, ChatGPT recommended different sensors depending on their use case (i.e., arcade-quality, DIY (budget), and DIY (premium))

After asking ChatGPT the question of: How can we implement computer vision into a dance rhythm game using a camera?, important information such as "core components" were given which included a camera, Computer Vision models such as MediaPipe, OpenPose, and Movenet were given, as well as how to implement these aspects into the game itself.[23]

Furthermore, ChatGPT gave us additional help as to how to the game should flow when implementing the Computer Vision models with the example of loading a song with a predefined chart, have the camera be able to detect player poses, and during each beat window, give a score depending on the threshold set, and have that score be reflected and shown on screen. This gives an idea of what we ourselves could use for the game implementation of the computer vision side.[23]

ChatGPT gave us an extensive and thorough explanation as to what we could add for the rhythm game synchronization aspect of the game. It gave us some steps as to how to add a music track, set the BPM for the song, add a timer, and have an on beat function. Furthermore what ChatGPT gave us an idea with , is thinking of how to implement with the game as a whole having a mapping of the series of arrows put onto the game. Using a .beatmap file or a JSON file would be most helpful with JSON being easier to do using a script to create a JSON file after pressing a series of buttons.

5.3 Google Gemini

Google Gemini gave us a more indepth look as to why game engines would be the best resource to use when creating a rhythm game. Google Gemini explains that game engines handle much of the work in regards to audio synchronization, inputs, and have communities that contain many forums, tutorials, and other useful tools for creating games. The game engines that Google Gemini recommended were similar to those of ChatGPT, including Godot and Unity, however did not mention scratch but rather mentioned GameMaker Studio. [25]

Google Gemini, much like ChatGPT, listed a few MCU options with pros and cons. It listed down each MCU's key features. Additionally, Gemini also listed down key considerations for our project. In the end, Gemini recommended Arduino Leonardo or Adafruit ItsyBitsy M0/M4.

Moreover, this time around, after searching our dance pad sensor selection question, we were not provided a pros and cons list. Instead, the best options are explained as to why they are best for certain cases - listing their key features and things to consider as we're deciding (e.g. weaknesses of the sensor). After listing the best options, it also listed less common and emerging options just to provide us more options. While Gemini was a lot more informative with this approach, it did not provide enough comparison between the different types, more specifically, a comparison table could've provided more clarity to what has been stated. In the end, Gemini recommended FSRs for both arcade-grade and DIY dance pads.

Similarly to ChatGPT, Google Gemini also gave us the idea to use MediaPipe or OpenPose for computer vision libraries that would be useful for camera integration, furthermore, Google Gemini also gave us an additional source of YOLO that also can be manipulated and changed to be able to include pose detection. Google Gemini then goes into how to obtain the data and process it using the camera's input and processing the information frame by frame into the computer vision library of choice. [25]

On the aspect of implementation with the rhythm game itself, Google Gemini additionally gave us information as to how to implement matching the pose to the given reference that we give it. There are many examples that were provided such as:

"Euclidean Distance", "Cosine Similarity", "Dynamic Time Warping" or through "Machine Learning/Deep Learning" [] For the game logic side, Google Gemini recommended having rhythm synchronization as well as visual feedback. We believe this can be done using some on screen text or prompt depending on the timing showing how well they performed the certain pose when told. The way to implement the pose timing is by having a silhouette of the pose itself on screen to indicate to the user which pose will be the one to pop up which thankfully was an idea that Google Gemini itself gave us as an option to use.

With Google Gemini, the benefits of using it to ask the question about rhythm synchronization, was that it gave us the core concepts of what we would need to include for rhythm synchronization to be accomplished in Godot. The concepts that are included are:

- Audio Playback
- Latency Compensation
- BPM
- Event Scheduling
- Conductor System

What Gemini also gives us are ways to implement each of these aspects. For example adding the Audio Stream Player, creating the conductor system, even how to add a rhythm notifier asset from the Godot asset store. Each of these aspects that Gemini gives us, helps us understand how to implement them and what process to complete it.

5.4 Microsoft Copilot

Microsoft Copilot was not a very useful tool as it gave similar information to ChatGPT and Google Gemini in regards to which game engines to use and had no deviation in regards to that information. Using a no-code design would not be useful for us and we would not be accessing the web for our project thus both section 2 and section 3 do not apply to our project.[24]

Microsoft gave options much like ChatGPT and Gemini, however, instead of listing pros and cons, it made a comparison table of each MCU's features (e.g., USB support, GPIO pins, etc.). It summarized that the given options had LED driver compatibility and dance pad input support, as well as other information to keep in mind regarding them. In the end, Microsoft copilot did not recommend a best-of-the-best MCU, instead just shared all options as the best for the design.

For our dance pad sensor question, Microsoft copilot was unique in that it provided actual products from actual shopping websites, although they were not useful at all. For instance, Microsoft sent product details on a whole functioning dance pad, and corner panel mounts. The question was to find the best sensors, yet it provided actual dance pads that already exist. However, afterwards, Microsoft copilot does go over the different types, but only the smallest amount of key features, though very generic and hard to understand. Microsoft copilot lacked in explaining meaningful information. In the end, Microsoft copilot recommended conductive plate sensors or Chinese arcade dance sensors for arcade-like feel, FSRs for customization and tech integration, or 3D printed spring sensors for an affordable and creative build.

Microsoft Pilot was not very useful after using both ChatGPT and Google Gemini as it gave very similar information on the topic. However, the one positive aspect of using Microsoft Pilot was the chart that it gave.

Table 5.1 *Microsoft copilot Computer Vision Table*

Tool	Use Case	Why It Works Well
MediaPipe	Fast pose detection	Lightweight & cross-platform
Unity + Barracuda	Game dev + ML model integration	Familiar workflow, compatible with C#
OpenCV	Image processing & camera control	Open-source, powerful CV toolkit

This gives very insightful information based on what we need in terms of computer vision. Whether that be to use MediaPipe, Unity and a machine learning model called Barracuda or another type of computer vision model, or using OpenCV. Each of these options have their strengths and weaknesses.

MediaPipe for example has fast pose detection and is lightweight however there could be the possibility that it is not as accurate as other computer vision algorithms like OpenCV. OpenCV is open-source and has the possibility of Image Processing and Camera Control so it is another source we can research to see if it would be the right fit for us.

Microsoft Pilot gives us something very similar to both Gemini and ChatGPT that recommends us to use an AudioStreamPlayer, create a JSON beat map, and have some sort of timer that allows for calibration for the timing.

Furthermore Pilot gives us a chart of different techniques as to how to make the project better and more efficient.

Table 5.2 *Microsoft copilot Rhythm Synchronization Table*

Technique	Benefit
Audio latency compensation	Adjust for input delay on different devices
MIDI or beat detection tools	Automate beatmap creation from music
Godot plugin ecosystem	Explore tools like BeatDetector or custom modules for audio analysis

5.5 DeepSeek

DeepSeek gave us the most amount of information regarding the game engine and the rhythm game. It recommended Unity and Godot as the top 2 choices for deciding which game engine to use, however it also included Unreal Engine which is a common game engine that is used. Furthermore DeepSeek also gave us insightful tips as to how to proceed with the game design of the rhythm game itself. These tips include starting small and afterwards adding more features, following online tutorials for the game engine of choice, and also attempting prototyping with different aspects that will be implemented.[26] DeepSeek was the most helpful in this case and had a positive outcome on how we continued with the project.

DeepSeek was almost similar to that of ChatGPT and Gemini's response, in that it not only listed pros and cons of each MCU, but also what they're best used for. However, DeepSeek did not list enough information explaining each MCU, nor do they use a comparison table for better clarity. Not enough numerical specifications are mentioned, leaving us wondering what else to keep note of like GPIO pins, or ease of assembly. In the end, DeepSeek recommended Raspberry Pi Pico for its best balance of price and features, or Arduino Pro Micro for its ease of use and plug-and-play USB functionality.

Regarding the dance pad sensor question, DeepSeek was almost exactly like ChatGPT. DeepSeek listed each of its best options and their pros and cons. However, much like ChatGPT, the listed pros and cons or additional information were lacking and only provided as a base for our research. In the end, DeepSeek mainly recommended FSRs, whether it be DIY using conductive material and Velostat, or the actual hard component of FSRs.

When it comes to the question of how to implement computer vision to our project with a camera, Deepseek, similar to ChatGPT and Copilot, gave us ways to implement pose detection using computer vision models such as MediaPipe or OpenPose, using keypoints to track movement joints, and have a coordinate system with those joints detected. Furthermore Deepseek went into more depth as to how to continue with the project through tracking, rhythm synchronization, scoring and validations, game interface, and even implementations with how to accomplish our goal. What is good about deepseeks answer was how it was able to explain the different aspects that we should take into consideration without giving us the way to do it and have it work it ourselves.

With regards to the question “How can we implement rhythm synchronization into a Godot project?” Deepseek had a different approach as to how to help us with this question specifically. Deepseek had the idea of telling us exactly what it is that we need to implement in our game project in terms of scripts as well as the different settings to change in our project settings. This, although helpful, wasn't what we were looking for, but more so ideas as to how we can accomplish our goals and we were looking for more ideas that could work and not the implementation done for us.

5.6 Conclusion

Using these Artificial Intelligence systems gave us a lot of information as to where we can start on the software side of creating the rhythm game, as well as the hardware side in terms of finding the best MCUs and dance pad sensors for our design. All four engines agreed that using a game engine is the most beginner-friendly and effective approach for a project of this scale. The most commonly recommended engines were Godot, Unity, and Unreal Engine, with Godot especially noted for being lightweight, open-source, and beginner-friendly with strong 2D support.

For MCU selection, there was a consistent set of recommendations across engines. ChatGPT suggested Teensy 4.0, Arduino Leonardo, Raspberry Pi Pico, and STM32 boards, highlighting USB HID support and LED driver compatibility. Gemini emphasized Arduino Leonardo and Adafruit ItsyBitsy M0/M4, while also listing key considerations like GPIO availability and ease of development. Microsoft copilot provided a feature comparison table without a definitive recommendation but confirmed all listed options were suitable. DeepSeek leaned toward the Raspberry Pi Pico for value and features or Arduino Pro Micro for ease of use and plug-and-play functionality.

For dance pad sensors, most engines leaned toward FSRs for both arcade-grade and DIY builds. ChatGPT and DeepSeek offered structured pros and cons and discussed budget vs. premium DIY builds. Gemini explained why certain sensors were ideal and even mentioned less common alternatives but lacked side-by-side comparison. Microsoft copilot stood out for mentioning physical products (though some were unrelated), and ultimately recommended FSRs, conductive plate sensors, and 3D-printed spring sensors, offering a broad range of creative solutions.

AI tools were instrumental in our project's development. For computer vision, Google Gemini helped us weigh the benefits of MediaPipe (lightweight, fast) against OpenCV (open-source, camera control) and suggested implementation ideas like pose indicators. For our Godot-based game, LLMs provided a framework for rhythm synchronization, with ChatGPT suggesting core functions and Gemini recommending visual feedback. Ultimately, this guidance provided a strong foundation, helping us efficiently choose the right game engine, MCUs, and sensor technology for our 9-directional dance pad.

For implementing computer vision, the majority recommended using MediaPipe or OpenCV as our computer vision model of choice. Additionally they gave us more information as to how to apply computer vision to the game. Whether that be through having silhouettes or some type of indicator to show that a pose is popping up. The Large Language Model of Google Gemini was extremely helpful when giving us a chart of the different computer vision models to use by explaining how MediaPipe is lightweight and has fast pose detections while OpenCV is open source and has good camera controls. It helped us have an idea of which ones can be used and what we should research more on.

For implementing rhythm synchronization the Large Language Models were very useful in helping us have a start and think about how to implement the synchronization. Due to this point having chosen Godot as our game engine of choice. The models gave us insight as to how to implement the synchronization by telling us various aspects that we must keep in mind when creating this. For example, ChatGPT recommended to add a music track, set the BPM for the song, add a timer, and have an on beat function when creating the synchronization.[] Google Gemini further expands on this by explaining how some visualization of synchronization existing would improve the quality of the project.

In the end, these AI tools provided a strong foundation for choosing not only the right game engine, but also the most suitable MCUs and sensor technologies for implementing our 9-directional rhythm game dance pad, helping us focus our research and development efforts effectively. It additionally helped us decided the best course of action in regards to both how to use computer vision for our project and how to implement rhythm synchronization which is integral to any rhythm game.

6. Hardware Design

6.1 Dance Pad Master Controller Board

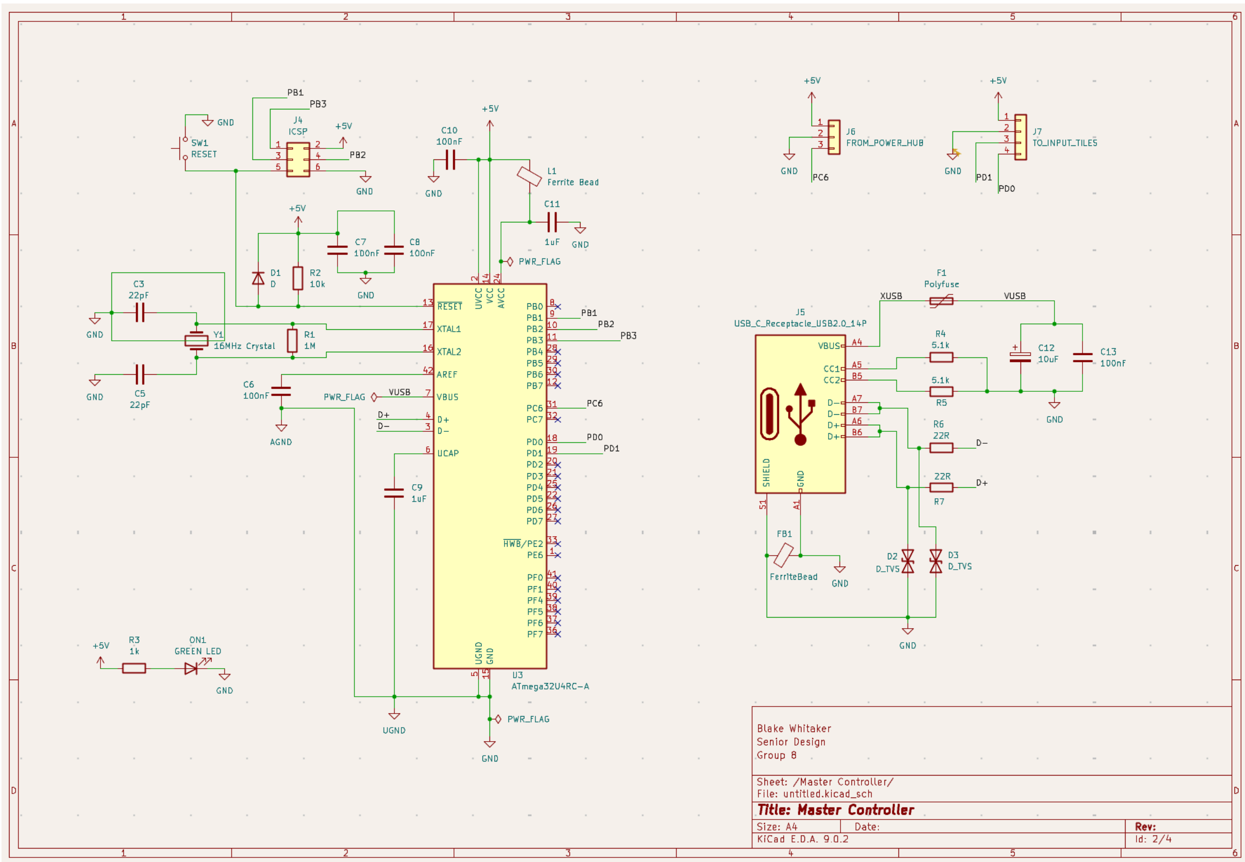


Figure 6.1 *Dance Pad Master Controller Board Schematic*

Core Functionality

This PCB is designed to be the “Master Controller” for the 9-tile dance pad game. In this modular design, this board's primary purpose is to act as the central brain or manager. It communicates with the host PC via USB, receives power from the Power Hub, and manages all nine smart Tile Boards over an I2C communication bus.

Circuit Breakdown

The heart of the board is an ATmega32U4RC-A microcontroller, designated as U1. This chip was chosen specifically because it has native USB capabilities, allowing it to easily communicate with a PC as a game controller without needing extra chips. A 16 MHz crystal oscillator, Y1, along with its two 22pF loading capacitors, C1 and C2, and a 1MΩ parallel resistor, R1, provides a precise and stable clock signal, which is essential for reliable USB communication. A simple pushbutton, SW1, is connected to the RESET pin, along with a 10kΩ pull-up resistor, R4, and a protection diode, D1, to allow for a manual reset during testing and development. A 6-pin header, J4, serves as the In-Circuit Serial Programming port, with its sole purpose being to allow the initial programming of the Arduino bootloader onto the fresh ATmega32U4 chip.

The USB interface section handles the data connection to the host computer and is designed to be very robust. A USB-C Receptacle, J2, provides the physical connection. This connection is protected by a 500mA resettable polyfuse, F1, which prevents the board from drawing too much current and damaging the computer's USB port. ESD protection diodes, D2 and D3, are placed on the D+ and D- data lines to guard the microcontroller against static shocks. Two 5.1kΩ pull-down resistors, R2 and R3, are connected to the CC1 and CC2 pins; this is a critical part of the USB-C specification that identifies the board as a device to the host computer. A ferrite bead, FB1, filters noise on the USB cable's shield connection.

The board receives its power from the external “Power Hub” board and carefully filters it for stable operation. The board is powered by a stable +5V and GND supplied via the 3-pin connector J7. To ensure clean power for the microcontroller's internal analog components, the AVCC pin is filtered through a ferrite bead, L1, and a 1µF capacitor, C3, which creates an LC low-pass filter to isolate it from noise. Several 100nF capacitors are placed near the chip's various power pins to filter out high-frequency noise and ensure stable operation. PWR_FLAG symbols are directives for the design software, confirming that the +5V, GND, AGND, and VUSB nets are intentionally powered, which resolves common electrical rule check errors.

This board serves as the central hub for all external connections. The 3-pin “Power Hub” Link connector, J7, receives +5V and GND to power this entire board, and it sends the PWM signal from pin PC6 to control the TV's IR illumination. The I2C Bus Output, J3, is a new, critical 4-pin connector for the modular design that serves as the starting point for the bus that connects to all nine “Input Tile” boards. It provides +5V and GND to power the entire chain of tiles and connects to the ATmega32U4's hardware I2C pins, PD1 for SDA and PD0 for SCL. A simple green "ON" LED, connected via a 1k resistor, provides a quick visual confirmation that the board is powered on.

6.2 Power Hub Board

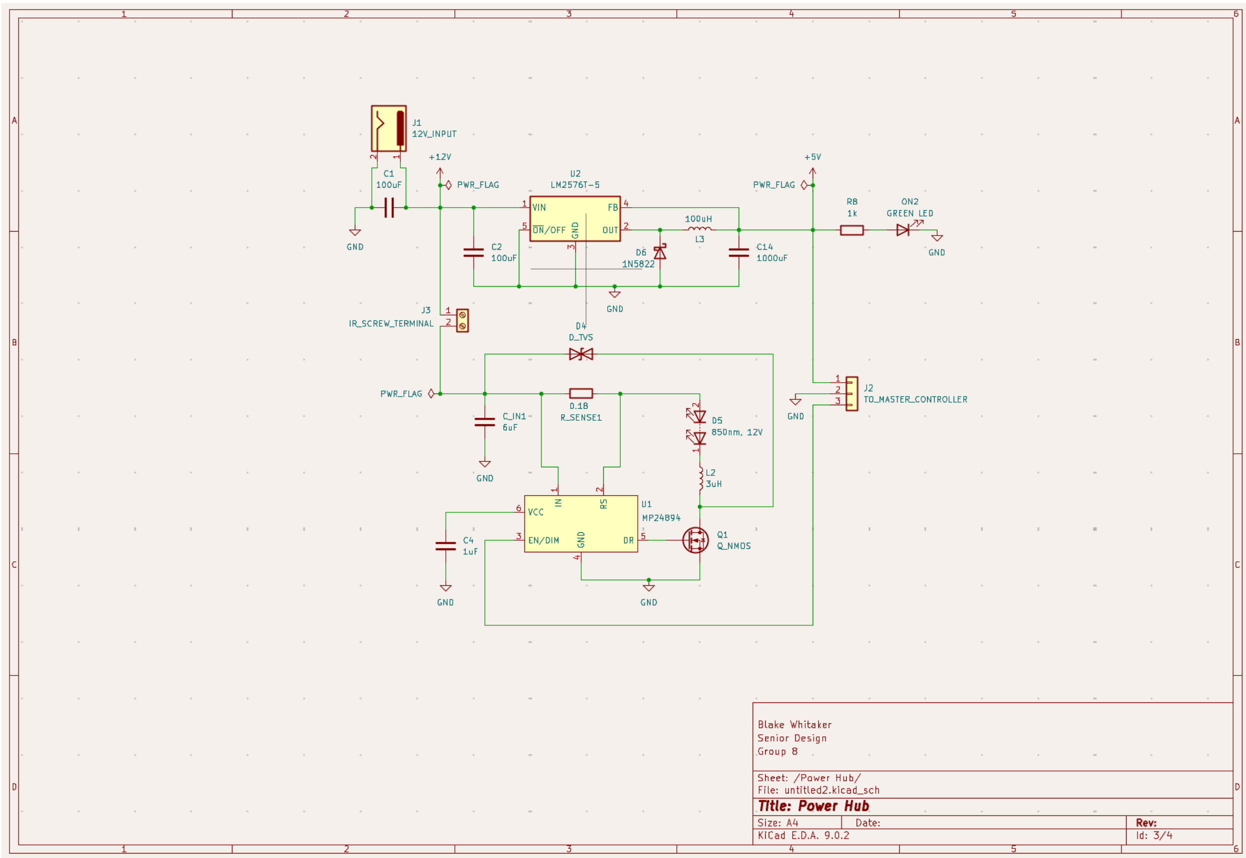


Figure 6.2 *Power Hub Board Schematic*

Core Functionality

This PCB is designed to be the central "Power Hub" and high-current driver for our dance pad project. Its primary purpose is to take a single 12V input and safely distribute power to all other parts of the system, while also controlling the high-power IR LEDs for the computer vision system.

Circuit Breakdown

The circuit's main power input is a standard DC Barrel Jack, J1, which accepts 12V DC from an external wall adapter. A 100µF capacitor, C1, is placed directly across the input to smooth out the incoming 12V supply and filter out any low-frequency noise or ripple from the power brick, providing a more stable input for the rest of the circuits.

This board creates the stable 5V needed to run the "Master Controller" board. This is accomplished by U2, an LM2576T-5, which is a highly efficient 5V switching buck regulator. It takes the 12V input and steps it down to a constant 5V output. A switching regulator was chosen over a linear one because it is far more efficient and generates significantly less heat. The circuit includes an input capacitor C2, a 1N5822 Schottky diode D1, a 100μH inductor L1, and a large 1000μF output capacitor C3. These components are required by the LM2576's datasheet to form the complete buck converter circuit. The regulated +5V is sent to the “Master Controller” board via the 3-pin connector J2. A green "ON" LED, powered through a 1kΩ resistor R34, provides a simple visual confirmation that this 5V supply is active.

The most advanced part of the board is the constant-current IR LED driver. The driver IC U1, an MP24894, is the brain of this circuit. It receives a PWM signal from the “Master Controller” and uses it to control the brightness of the LEDs. The MP24894 sends a precise drive signal from its DR pin to the gate of an external N-Channel MOSFET, Q1, which acts as the main switch for the high-power LEDs. The IC constantly monitors the current flowing through the LEDs by measuring the voltage across the sense resistor, R_SENSE, and adjusts the switching of the MOSFET to keep the current stable. A 2-pin screw terminal, J3, provides a secure physical connection point for the wires from the external 12V IR LED strip.

The 3-pin connector, J2, is the vital link between the “Power Hub” and the “Master Controller”. Pin 1 sends the regulated +5V power to the controller board. Pin 2 provides the common ground connection between the two boards. Pin 3 receives the PWM control signal from the controller board, which is then sent to the EN/DIM pin of the MP24894 driver to control the IR LED brightness. In summary, this board acts as a robust and intelligent power distribution center, ensuring both the controller and the high-power LEDs get the clean, regulated power they need to operate reliably.

6.3 Input Tiles Board (x9)

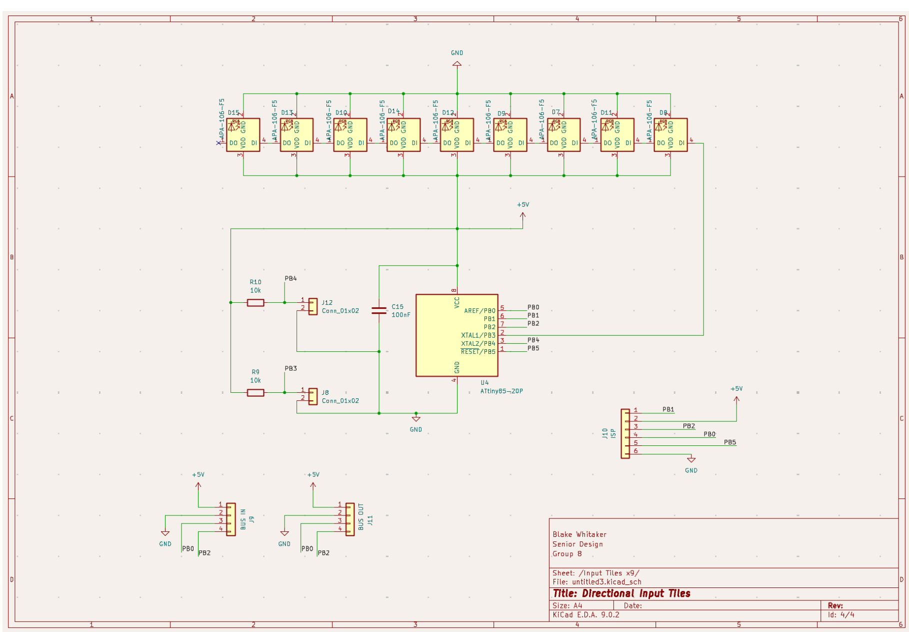


Figure 6.3 *Input Tiles Board Schematic*

Core Functionality

Nine of these identical boards will be manufactured, each of these custom PCBs are designed to be a modular "smart tile" for the 9-tile dance pad game. with each one responsible for the sensing and lighting of a single tile.

Circuit Breakdown

The core of the Tile Board is an ATtiny85 microcontroller, designated as U4. This small, 8-pin chip acts as the local brain for the tile. Its primary functions are to continuously read the pressure sensors, control the on-board aesthetic LEDs, and communicate with the “Master Controller” board over an I2C bus. A 100nF capacitor, C15, is placed across the microcontroller's power and ground pins to decouple the power supply, ensuring stable operation by filtering out high-frequency noise.

The board features two independent analog sensor circuits to read from two separate off-board Force-Sensitive Resistors. The first FSR connects to the board via the 2-pin connector J7, where it forms a voltage divider with the 10kΩ resistor R7. The resulting analog voltage is read by pin PB4 of the microcontroller. The second sensor circuit is identical, using the connector J8 and the 10kΩ resistor R9, with its output read by pin PB3 of the microcontroller. This dual-sensor design provides redundancy and allows for more complex pressure sensing logic in the software.

Aesthetic and interactive lighting is provided by a chain of nine addressable APA-106-F5 RGB LEDs, labeled D7 through D15. These LEDs are powered directly by the board's 5V and GND nets. The data signal originates from pin PB1 of the microcontroller and is daisy-chained from the Data Out of one LED to the Data In of the next. The Data Out pin of the final LED in the chain, D13, is left unconnected. This configuration allows the local ATtiny85 to have full, individual control over the color and brightness of every LED on its tile.

Communication and power are handled via a 4-wire bus system, facilitated by two 4-pin connectors, J9 for "Bus IN" and J11 for "Bus OUT". These connectors are wired in parallel to allow the bus to be daisy-chained from one tile to the next. The bus provides the +5V and GND to power the entire board. It also carries the two I2C communication lines. The SDA line is connected to pin PB0 of the microcontroller, and the SCL line is connected to pin PB2. This allows the board to act as an I2C slave, responding to commands from the “Master Controller”.

Finally, a 6-pin header, J10, serves as the In-System Programming port. This connector is used exclusively for loading the initial firmware onto the ATtiny85 chip after assembly. It connects to the VCC, GND, RESET, MOSI, MISO, and SCK pins of the microcontroller, allowing an external programmer to interface with the chip.

6.4 System-Level Hardware Integration

While each of the three custom PCBs serves a distinct purpose, the true functionality of the STEPS hardware emerges from their carefully orchestrated interaction. The system architecture is a distributed network where power, control signals, and sensor data flow in a coordinated manner between the “Power Hub”, the “Master Controller”, and the nine “Input Tiles”. This modular approach is designed to create a robust and scalable system.

6.4.1 Power Distribution and Regulation

The “Power Hub” board is the entire system's electrical focal point. A 12V DC signal from an external power brick enters through the barrel jack (J1). This board immediately splits the power into two main pathways, a High-Power Path (12V) is the raw 12V input is routed directly to the high-current side of the board to supply the MP24894 constant-current LED driver circuit. This path is designed to handle the significant current draw of the external infrared LED strips that border the external display used for the computer vision system. The other path is a Low-Power Logic Path (5V) where the 12V input is also fed into an LM2576T-5.0 switching buck regulator (U2). This highly efficient regulator steps the voltage down to a stable, clean +5V. This 5V rail is essential for all the logic-level components in the entire dance pad.

From the Power Hub, this regulated 5V power, along with a common ground, is sent to the Master Controller Board via a 3-pin JST connector (J2 on the Power Hub, J7 on the Master). The Master Controller then becomes the distribution point for the rest of the system. It uses this 5V to power its own ATmega32U4 microcontroller and then passes the 5V and GND lines out through its 4-pin I2C bus connector (J3). This single connector begins a daisy chain that delivers power to all nine Input Tile Boards, ensuring every ATtiny85 microcontroller and its associated RGB LEDs receive stable power.

6.4.2 Control and Data Flow

The flow of information is managed by the Master Controller, which acts as the central nervous system. It communicates both with the PC and with its subordinate Tile Boards.

Each of the nine Input Tile Boards operates as an I2C slave device with a unique, pre-programmed address. The ATtiny85 on each tile continuously monitors the voltage from its two Force-Sensitive Resistors (FSRs). The “Master Controller” acts as the master to constantly poll each of the nine tile addresses in a rapid loop. When a tile receives a request from the master, it sends back a small data packet containing its current sensor readings. This distributed processing approach, where each tile manages its own sensing, prevents the Master Controller from being burdened with nine separate analog-to-digital conversions.

When the Master Controller receives data from a tile indicating a step has occurred (FSR pressure has crossed a given threshold), its firmware processes this event. The ATmega32U4 then uses its native USB capabilities to format the step data into a standard Human Interface Device (HID) report which emulates a generic game controller or keyboard input. This report is sent to the host PC over the USB-C connection. The game software on the PC receives this as a simple button press, requiring no custom drivers and ensuring maximum compatibility.

Communication on the I2C bus is bidirectional. The game can send commands back to the Master Controller, which then relays these commands to the appropriate Input Tile Board. For example, upon a "Perfect" step, the master can send a command to Tile #5 to trigger a flashing green light effect for example. The ATtiny85 on that tile receives the command and drives its local array of APA-106 RGB LEDs to produce the desired visual feedback.

6.4.3 Vision System Illumination Control

The control loop for the computer vision system's IR illumination demonstrates the full integration of all three boards. First, the game logic on the PC determines that the IR LEDs need to be activated. Second, the Master Controller's ATmega32U4 generates a Pulse-Width Modulation (PWM) signal on pin PC6. The duty cycle of this signal corresponds to the desired brightness of the LEDs. Third, this PWM signal travels from the Master Controller to the Power Hub Board via the 3-pin JST link. Finally, on the Power Hub, the MP24894 LED driver IC (U1) receives this PWM signal on its EN/DIM pin. It interprets the duty cycle and drives the external high-power MOSFET (Q1) accordingly, delivering a precise, constant current to the 12V IR LED strips.

In summary, the hardware operates as a cohesive, hierarchical system. The Power Hub provides conditioned power, the Input Tiles act as localized sensor nodes, and the Master Controller serves as the central aggregator and communicator, seamlessly bridging the physical actions of the player with the software of the game.

6.5 Optical Imaging System Design

The optical subsystem is designed to reliably capture and analyze a player's full-body movements in real time, enabling the system’s pose-based “Style Score” feature. This subsystem integrates a global shutter camera, wide-angle lens, and an optimized near-infrared (NIR) LED illumination system to provide robust visual tracking across the entire play area.

The vision system must cover the 2.9m x 2.9m tracking area at a typical player distance of 1.83m, maintaining a horizontal field of view of approximately 93°. This ensures consistent landmark detection and minimizes distortion even at the edges of the field.

A time-multiplexed 850nm near-infrared LED system was selected to provide uniform, comfortable illumination that is invisible or only partially visible to the player but reliably detected by the camera. This design minimizes distractions while maintaining sufficient brightness for accurate MediaPipe landmark tracking.

In alignment with the project's broader objectives stated in Chapter 2, the optical subsystem emphasizes consistent player coverage, high image clarity, safe illumination, and efficient integration with the vision-based scoring system. The imaging system is designed to deliver high spatial resolution, wide field of view, and low-latency image capture that supports real-time pose estimation. After evaluating multiple options, the SVPRO AR0234 camera module was selected. This camera was selected based on its technical specifications, including a 2.3MP resolution (1920 * 1200), 3µm pixel pitch, and frame rates up to 60 FPS over USB 3.0. The global shutter feature helps reduce motion artifacts during rapid gameplay.

Given the player tracking area dimensions are 2.9m x 2.9m and the working distance is 1.83m, the optical system must maintain a horizontal field of view of approximately 93° to reliably capture the entire play area with minimal occlusions. Commercial lenses with longer focal lengths or narrow fields of view are impractical, since they would have required either excessive installation height or more aggressive mounting angles to fully capture the 2.9 m x 2.9m play area. To avoid these challenges, the system instead uses a commercially available short focal length M12 lens that is approximately 3mm. It has a diagonal FOV between 100-114 degrees. The key engineering requirements for the optical subsystem are summarized in Table 6.1.

Table 6.1 *Engineering requirements for the optical imaging subsystem*.

Component	Parameter	Specification	Unit
Camera	a) Pixel Pitch b) Resolution c) Frame Rate	a) 3 b) 1920 x 1200 c) ≥ 60	a) µm b) pixels c) FPS
Lens (M12 3mm)	a) Focal Length b) Horizontal Field of View	a) 3 b) 93	a)mm b)degrees
Optical Subsystem	a) Brightness Uniformity b) Resolution at 6ft	a) ≥85 b) ≥ 3 (resolving ~ 1mm features)	a)% b) pixels/mm

As summarized in Table 6.1, the selected camera module and lens combination achieves a pixel pitch of 3 µm with a resolution of 1920 x 1200 pixels, is sufficient to meet the target of ≥ 3 pixels/mm needed for ~1mm feature detection at a 1.83m distance. The approximate 93° horizontal field of view ensures the full dance pad remains visible, while the geometric distortion and brightness uniformity specifications help preserve consistent landmark tracking across the entire field of play.

6.6 Illumination System Design

Consistent and uniform illumination is essential. In a dim arcade environment or under changing lighting conditions, the system still needs to have consistent illumination of the player. Therefore, relying on ambient light alone is insufficient for consistent pose estimation. To account for this, an active illumination system was designed to provide uniform coverage across the entire 2.9m x 2.9m player tracking area.

850nm near-infrared (NIR) LED strips were selected as the primary illumination source. Near-infrared wavelengths are partially invisible to the human eye, minimizing player distraction and discomfort while remaining fully detectable by the camera's image sensor, which does not have an IR-cut filter. This approach allows for reliable landmark visibility for MediaPipe while not compromising on user comfort.

To further optimize power consumption and thermal performance, a time-multiplexing strategy is being used. In this scheme, the dance pad area will be divided into multiple LED illumination zones, with only one zone activated per camera frame. Cycling rapidly at the camera's frame rate of 60 FPS, resulting in a complete zone cycle every 17ms. Because this frequency exceeds the human flicker fusion threshold, players and the camera module will still perceive the illumination as continuous.

The selected LED strips are standard 12 V SMD-based products, offering good cost efficiency and a wide choice of densities (typically 60-120 LEDs/m). Each strip is rated at approximately 14.4 W/m, with power budgets calculated to keep the current draw under 2A per zone. Table 6.2 summarizes the engineering requirements for the illumination system.

Table 6.2 *Engineering requirements for the illumination subsystem*

Component	Parameter	Specification	Unit
LED Illumination	a) Wavelength b) Brightness Uniformity c) Beam Angle d) Zone Switch Time e) Power per Zone	a) 850 b) ≥85 c) ≥ 120 d) ≤ 17 e) ≤ 18	a) nm b) % c) degrees d) ms e) W
Illumination System	a) Coverage Area b) Player Distance	a) 2.9 * 2.9 b) 1.83	a) m b) m

As summarized in Table 6.2, the 850 nm LED strips are designed to deliver at least 85% uniform brightness across the entire field of view, with a wide beam angle of ≥ 120° to minimize shadows. Time-multiplexing with a zone switch time of ≤ 17 ms reduces the total system load while maintaining a seamless visual.

Safety and photobiological compliance were also considered. IEC 62471 guidelines were reviewed, and the selected 850 nm LED strips fall within Risk Group 1 at specified radiant exposure and player distance. Although typically arcade lighting is rarely completely dark, its variability, color saturation, and moving shadows make it unreliable for consistent pose detection. Therefore, the NIR illumination system is necessary to establish a stable, uniform baseline at the player's distance of 1.83m. Providing radiant intensity equivalent to at least 300 lux of visible illumination for consistent landmark detection. Since the dance pad is intended to be portable for use in home settings, the lighting conditions there could be even more unpredictable, ranging from bright windows to dim living rooms. Including a controlled illumination system ensures consistent MediaPipe performance regardless of whether the system is deployed in a commercial arcade, dorm room, or any other household living space.

The camera module will be mounted at approximately 1.5m height with a downward angle of about 15-20° to capture the full play area while minimizing occlusions from player movement. The LED strips are currently planned to be mounted around the monitor frame to provide even IR coverage while avoiding direct glare into the player's eyes. However, other LED mounting configurations will also be tested in SD2 to determine the most effective combination of coverage uniformity, ease of integration, and player comfort. The final arrangement will be chosen based on uniformity measurements and practical field trails to ensure consistent landmark detection performance.

6.7 Optical System Testing and Verification

The optical and illumination subsystems will be tested to confirm they meet the design requirements described above. Testing will focus on verifying the horizontal field of view covers the entire dance pad area, confirming the spatial resolution supports ~ 1 mm feature detection, measuring illumination uniformity across the player area, ensuring time- multiplexing achieves seamless coverage without perceptible flicker, and confirming the system achieves at least 300 lux equivalent at the player's distance.

Initial testing will involve capturing images of a printed calibration checkerboard to measure geometric accuracy, including field of view, distortion, and alignment[8]. Separately, a uniform reflectivity target such as a matte white board will be used to verify even illumination across the play area. Pixel intensity measurements from these images will be analyzed to assess both brightness uniformity and overall image quality. If any issues such as uneven lighting, distortion, or poor focus are identified, adjustments will be made to the lens focus, LED placement, or camera alignment prior to full integration.

7. Software Design

7.1 Input/Output

The primary method of input for the rhythm game is the custom-built dance pad, which functions similarly to a keyboard from the perspective of the operating system. During gameplay, the game continuously polls for input every frame, much like a traditional PC game responds to keyboard input. Our custom PCB includes a microcontroller (MCU) responsible for interpreting signals from force sensing resistors (FSRs) embedded in each panel of the dance pad. These signals are converted into digital keypresses and transmitted to the host computer over a USB connection. As far as the PC is concerned, the dance pad appears as a standard USB keyboard, with each directional panel mapped to a specific keycode.

Table 7.1 *Game Arrow Mapping*

Game Interpretation	Keyboard Keycode	Dance Pad Panel
□	**W**	□
□	**E**	□
□	**D**	□
□	**C**	□
□	**X**	□
□	**Z**	□
□	**A**	□
□	**Q**	□
⏺️	**S**	⏺️

The table above shows the different keyboard inputs that are able to be used especially for those who would like to import the game to a computer and want to use the computer's built-in keyboard or an external keyboard of their choosing. This allows for portability for the game itself.

This input abstraction allows for flexibility during development and testing. Although the dance pad is the intended method of interaction, players could also use a conventional keyboard to play the game. However, this alternative would lack the physicality and immersive experience that the dance pad is designed to provide.

The game maintains an internal representation of the current input state, tracking which directional inputs are actively being held and recording the precise timing of each press event. When a player steps on a panel corresponding to a direction (such as up, down, left, right, center, or any of the diagonals), the game updates its internal state to reflect that the direction is currently active. It also records the timestamp or frame number at which the input was first detected. This information is used for evaluating the accuracy of the player's timing during gameplay.

When the player lifts their foot and the input is released, the game updates the state to mark that direction as no longer active. This real-time management of directional states and associated timing data ensures that the game can accurately judge the player's inputs against the expected cues defined in the song chart. A diagram of this input and output flow is shown in Figure 7.1.

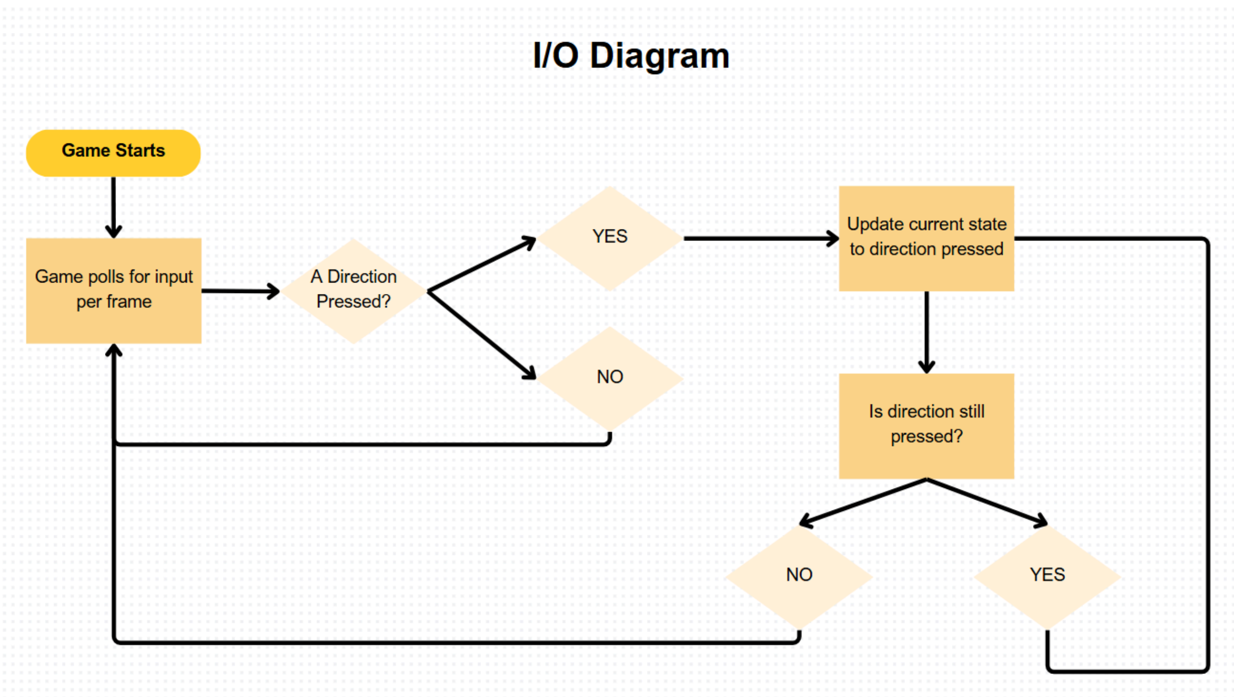


Figure 7.1 *Input/Output Flow Diagram*

7.2 State

The concept of state plays a central role in the software's logic. The input state is composed of several components, with each of the nine directional inputs represented as individual objects. These objects include a boolean indicating whether the input is currently held down, and if so, they also store the time or frame at which the press began. This enables precise tracking of how long a panel has been held and when the input occurred relative to the rhythm chart.

In addition to directional input, the system also tracks the player's current pose using the output from the computer vision subsystem. The pose state is updated every frame based on the latest keypoint and gesture analysis results. Each pose is identified by a label (such as “hands up” or “lean left”) and is associated with a confidence score and the time at which it was first detected. The pose state behaves similarly to directional input, allowing the game to evaluate whether the player is performing the correct pose at a given moment in the song.

The system also maintains a variable that stores the most recently changed input or pose, useful for debugging, scoring, and triggering visual effects. At the start of each song, the input state is initialized to a neutral condition, with no active inputs or poses. As gameplay progresses, the state evolves to reflect the player’s ongoing actions. The flow of this state update process is illustrated in Figure 7.2, which outlines how the game responds to input and pose events.

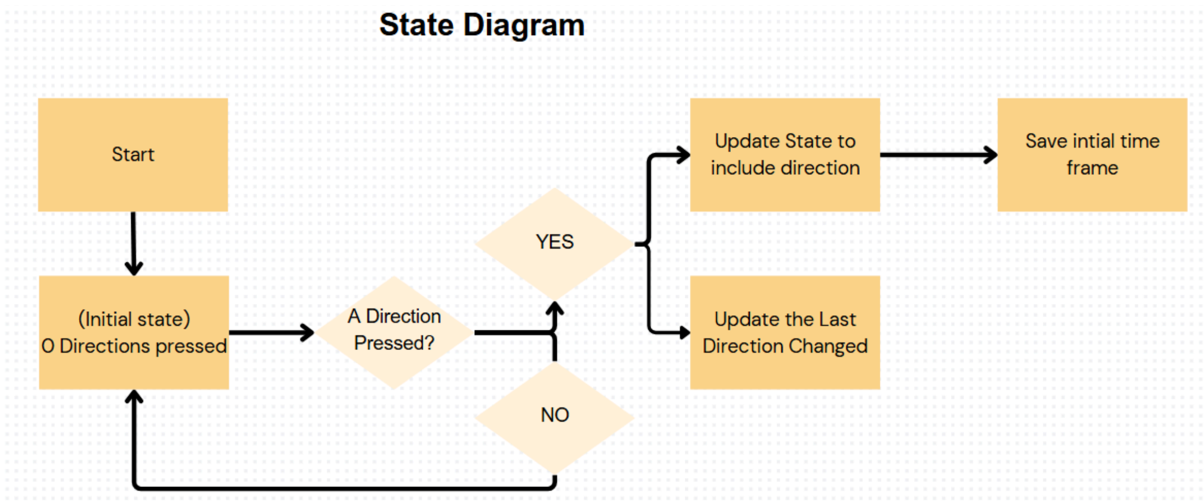


Figure 7.2 *Note State Flow Diagram*

7.3 Event Handling

Scoring in the rhythm game is driven by discrete gameplay events that are defined in the song chart. These events fall into two main categories: directional note events and pose-based gesture events. When the game encounters one of these events during playback, it checks the current input state to determine whether the player has correctly performed the expected action.

Each event is associated with a specific timestamp or frame number. At the moment an event is triggered, the game compares the expected input or pose to the actual state of the player, as recorded in the current frame and recent frame history. A timing window is applied to judge how closely the player’s action aligns with the event.

The judgment criteria follow a frame-based window around the expected frame of the event:

- If the input or pose is detected within ±3 frames, the judgment is marked as Perfect.
- If detected within ±5 frames, it is marked as Great.
- If within ±8 frames, it is considered Okay.
- If the correct input or pose is not detected within 8 frames of the event, it is scored as a Miss.

This scoring logic is applied consistently to both directional and pose events. However, additional tolerance may be applied to pose events to account for the inherent variability of real-time pose detection. Factors such as model inference delay, confidence thresholds, and temporal smoothing can introduce minor inconsistencies, so the system may offer a slightly wider scoring window for gestures without compromising the integrity of the gameplay.

By evaluating input and pose accuracy within tightly defined timing windows, the game can provide responsive feedback and fair scoring that rewards player precision while accommodating the occasional noise present in computer vision systems.

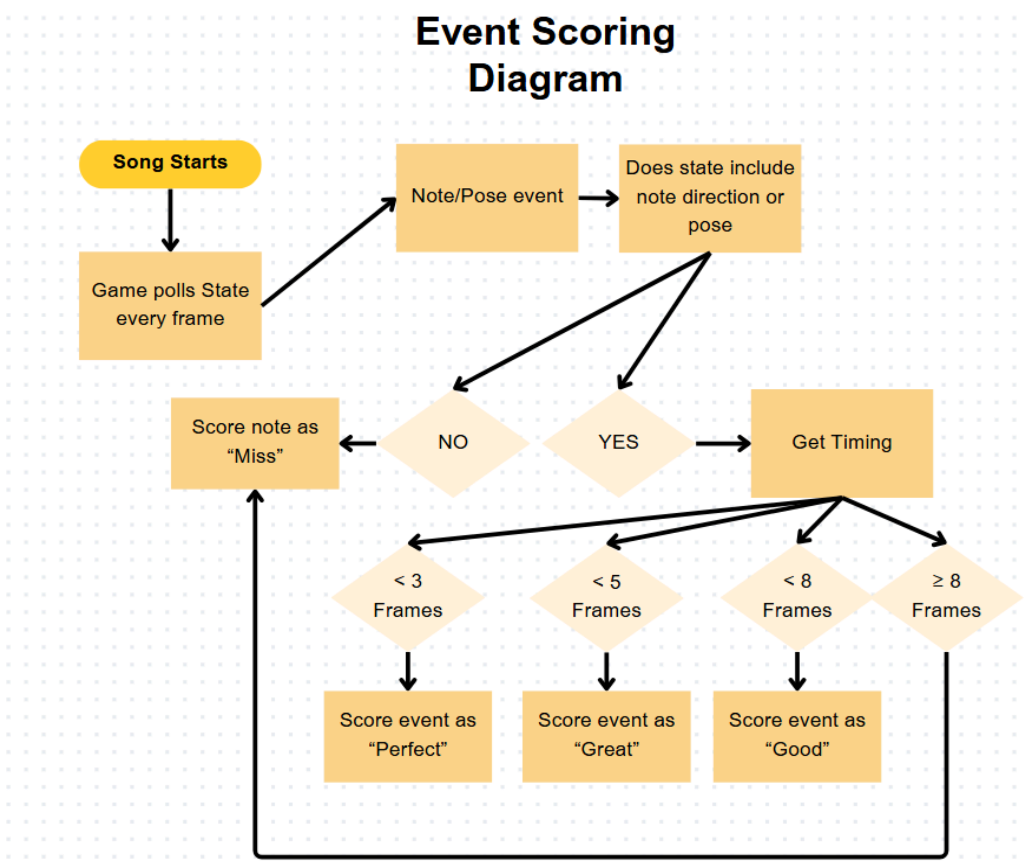


Figure 7.3 *Event Scoring Flowchart*

7.4 Pose Handling

Using the OpenCV and Mediapipe software, we first obtain the image sent by the camera module that will show the person. Afterwards our machine learning model activates that is what allows the person to be mapped using a series of dots as seen in *Figure 3.10*. We will then retrieve the series of dots location and with this information we can determine if a certain pose is being performed. If a certain pose is being performed and recognized, then the state will be updated to reflect that. However, if a pose is not being performed the state will be updated to show that there is no pose that is being performed.

Using this system there is a possibility due to artificial intelligence, that the state could be incorrectly updated, however with our system there is a 90% accuracy in correctly determining the correct state when performing these series of poses at certain portions of the game's design.

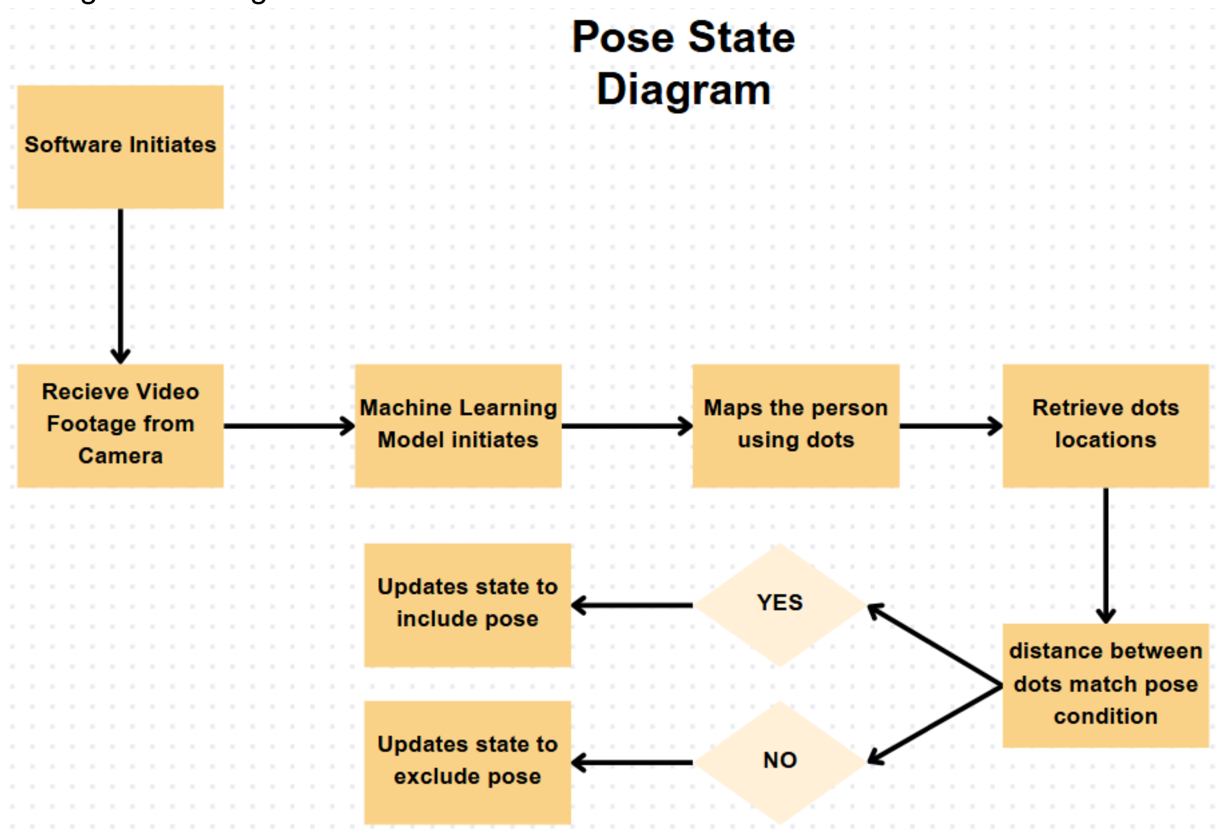


Figure 7.4 *Pose State Flow Diagram*

8. System Fabrication

The fabrication of our system hinges on the careful design and implementation of its custom Printed Circuit Boards (PCB)'s. The system's distributed architecture comprises three unique designs: a central "Master Controller", a high-current "Power Hub", and nine identical modular "Input Tiles". Each of which requires a robust and well-considered PCB layout. A proper layout is not merely about connecting components, but a critical engineering discipline that ensures signal integrity, power stability, thermal management, and manufacturability. The following will provide details on the layout philosophy and specific design choices for each of the three custom PCBs, adhering to industry standards such as IPC-2221A to translate the schematic designs into reliable, high-performance hardware.

8.1 PCB Layout Strategy

The core philosophy guiding the layout of all STEPS PCBs is one of partitioning and isolation. Each board is logically divided into functional blocks, such as power supply, microcontroller core, high-speed communication, and analog sensing. By physically grouping the components of each block and carefully managing the routing between them, we can minimize electromagnetic interference (EMI), prevent noisy digital circuits from corrupting sensitive analog signals, and ensure stable power distribution.

Some key principles applied across all designs include strategic component placement to minimize trace lengths, especially for high-frequency signals (like crystal oscillators and USB data lines) and sensitive analog signals. Trace width management calculated based on the IPC-2221A standard. High-current power lines are made significantly wider to handle the electrical load without overheating, while non-critical signal lines use a standard width. A solid ground plane is used on at least one layer of each PCB. This provides a low impedance return path for all signals, which is crucial for reducing noise, EMI, and aiding in thermal dissipation. Various decoupling and filtering capacitors are placed as close as possible to the power pins of every integrated circuit. This provides a local reservoir of charge to handle sudden current demands and shunts high-frequency noise to the ground plane before it can propagate through the system.

8.2 Master Control Board Layout

The Master Controller is the central nervous system of the dance pad. Its layout is optimized for reliable communication with both the host PC via USB and the nine "Input Tiles" via I2C.

The ATmega32U4 (U1), its 16 MHz crystal oscillator (Y1), and the crystal's loading capacitors (C1, C2) are placed centrally. The crystal is positioned immediately adjacent to the MCU's XTAL pins. This is the most critical placement on the board, as long traces to the crystal can act as antennas, introducing noise and causing clock instability, which would lead to catastrophic USB communication failure.

All components related to the USB connection are grouped together at the edge of the board for direct external access. The USB-C connector (J2), ESD protection diodes (D2, D3), and the 5.1kΩ configuration resistors (R2, R3) are tightly clustered. The differential data lines (D+ and D-) are routed as a pair with matched lengths and minimal distance to maintain their characteristic impedance and protect them from noise. The 500mA polyfuse (F1) is placed directly after the VBUS pin of the connector to protect the host PC's USB port from overcurrent events.

Power enters from the Power Hub via connector J7. To ensure the MCU's sensitive analog-to-digital converter (ADC) receives clean power, the AVCC pin is isolated from the main 5V rail by an LC filter network (ferrite bead L1 and capacitor C3). A series of 100nF ceramic decoupling capacitors are placed next to every VCC and AVCC pin on the microcontroller, providing essential high-frequency noise filtering.

The 4-pin connector (J3) that serves as the master output for the I2C bus is positioned to allow for a clean run of traces from the MCU's hardware SDA (PD1) and SCL (PD0) pins. This minimizes the length of the bus on the master board before it is sent out to the daisy-chained tiles.

8.3 Power Hub Board Layout

The Power Hub board manages a mixed-power environment, handling both a high-current 12V circuit for the IR LEDs and a low-current, regulated 5V supply for the system's logic. The primary layout consideration is the strict isolation of the high-power switching circuits from the sensitive 5V logic circuits.

The board is physically partitioned. Components for the 12V IR LED driver are on one side, and components for the 5V buck regulator are on the other. This prevents the high frequency switching noise from the LED driver from coupling into the clean 5V rail.

5V Switching Regulator (LM2576): All components associated with the buck regulator (U2, inductor L1, Schottky diode D1, and input/output capacitors C2/C3) are grouped tightly together, following the layout recommendations in the component's datasheet. Keeping the switching loop (from U2, through L1 and C3, and back to U2's ground) as small as possible is critical for efficiency and minimizing radiated EMI. The 5V output and 12V input traces are made very wide to handle the current without a significant voltage drop.

Constant-Current LED Driver (MP24894): The circuit for the LED driver is also a high-frequency switching regulator. The IC (U1), the main switching MOSFET (Q1), the sense resistor, and the flyback diode are all placed very close to one another. This minimizes the area of the high-current switching loop, which is essential for reducing

EMI. Large copper pours are used for the 12V and GND connections to the MOSFET to act as a heatsink, helping to dissipate the heat generated during high-current operation.

Grounding: A carefully designed ground plane is used. While a single ground plane is used for simplicity, the layout ensures that the return path for the noisy LED driver current does not flow underneath the sensitive 5V regulator circuitry. This is a form of "virtual" partitioning that maintains ground integrity while preventing noise coupling.

8.4 Input Tile(s) Layout

The nine “Input Tile” boards are the sensory and feedback nodes of the system. As they are identical and modular, their layout is optimized for easy assembly, reliable operation, and simple daisy-chaining.

The ATtiny85 microcontroller (U4) is placed centrally, with its 100nF decoupling capacitor (C15) located directly adjacent to its VCC and GND pins. The connectors for the Force-Sensing Resistors (J7, J8) and their corresponding voltage divider resistors (R7, R9) are placed close to the MCU's ADC input pins (PB3, PB4). This keeps the analog traces short, making them less susceptible to picking up digital noise from other parts of the board, which could lead to inaccurate pressure readings.

The nine addressable RGB LEDs (D7-D15) are laid out in a 3x3 grid that physically corresponds to their intended placement within the dance pad tile. The data line is routed sequentially from the MCU to the first LED's Data-In pin, and then from each LED's Data-Out to the next one's Data-In. The power traces feeding the LEDs are made sufficiently wide to handle the cumulative current draw of all nine LEDs when they are at full brightness.

The "Bus IN" (J9) and "Bus OUT" (J11) connectors are placed on opposite edges of the board. The VCC, GND, SDA, and SCL traces run directly across the board between these two connectors. This layout makes it mechanically simple to connect the tiles to one another in a chain using short, standardized cables, greatly simplifying the final assembly of the dance pad.

9. System Testing and Evaluation

9.1 Prototype Construction

Our prototype dance pad tile is constructed using a layered approach to ensure durability and functionality. The foundation is a solid piece of plywood, which serves as a rigid and stable base for all other components. On the underside of this base, small wooden blocks act as feet to elevate the tile and provide stability on the floor. A raised inner frame, made from cut pieces of acrylic, is built on top of the plywood base. This frame defines the active area of the tile. The Force-Sensitive Resistors are placed inside the area defined by this frame, with two FSRs laid horizontally in the images. Circular cutouts in the base layer are intended for routing the sensor wires cleanly out of the tile assembly. The top surface that the player steps on is a thick sheet of clear acrylic. This transparent top rests directly on the raised wooden frame. When a player steps on the tile, their force is transferred through the top sheet into the sturdy frame, which protects the sensors below from being crushed. The top sheet only needs to flex a minimal amount to activate the FSRs.

Figure 9.1 *Top view: 10x10 inches central acrylic tile, 12x12 inch w/ border, secured on wood base w/ standoff blocks*

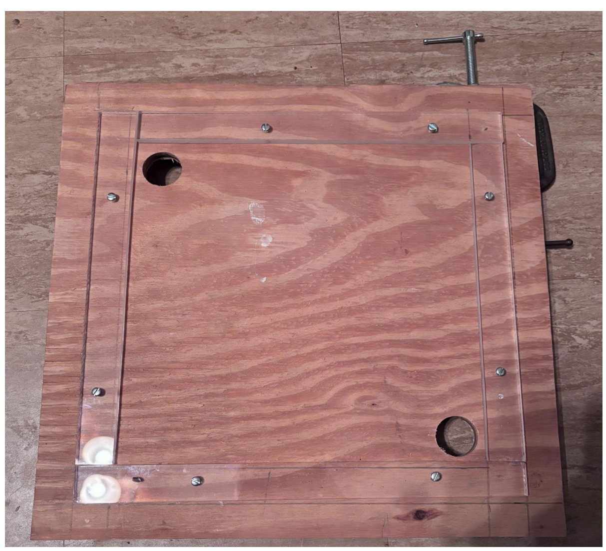


Figure 9.2 *Top view w/ 10x10 inch central tile removed showing electrical routing holes*

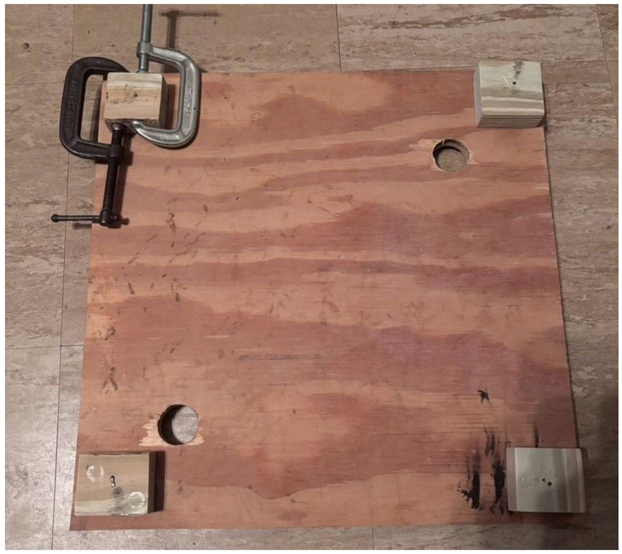


Figure 9.3 *Bottom view: standoff blocks to allow for easy cable routing*

For this prototype phase, the electronics are not on a custom PCB. Instead, the wires from the FSRs will be connected to a solder-less breadboard, which will then be connected to the Arduino Mega. To test the functionality of this prototype, we first built the voltage divider circuit on a solder-less breadboard by connecting a 10kΩ resistor between the 5V rail and a separate row. The two wires from one of the FSRs inside the prototype tile are then connected, with one wire going to the ground rail of the breadboard and the other connecting to the same row as the resistor. Next, we connected the breadboard to the Arduino Mega by running wires from the breadboard's 5V and ground rails to the corresponding 5V and ground pins on the Arduino. A signal wire is then run from the row where the FSR and resistor meet to the analog input pin A0 on the Arduino Mega.

Figure 9.4 *Top View: w/ Force Sensitive Resistors (FSR) placed under acrylic tile*

9.2 Hardware and Software Testing

After the physical connections were made, we connected via USB to the Arduino Mega and created a simple test program. This sketch was written to continuously read the analog value from the analog input pin A0 and print that value to the serial monitor. With the serial monitor open we observed the baseline readings with no pressure applied. Then, we proceeded to press on different areas of the tile with our hands and feet to observe how the values change. This process was used to help determine the best orientation for the FSRs to achieve maximum input coverage and gather the data needed to define a reliable software threshold for detecting a step.

9.3 Performance Evaluation

9.3.1 Hardware

For the FSR-based input panels, we experimented with multiple configurations to determine the most responsive and consistent placement strategy. Initial testing compared diagonal placements, single-sensor configurations, a two-FSR cross arrangement, 4 FSRs around the perimeter of each panel, pairs of FSRs more towards the center where the player would be, and pair of FSRs on opposite ends. Ultimately, we found that positioning two FSRs on opposite ends underneath each step panel provides the most reliable activation and minimizes the chances of unregistered steps due to uneven foot pressure. This FSR layout is shown in *Figure 9.5*. However, this configuration will require additional stress testing during SD2 to evaluate long-term reliability and edge responsiveness under repeated dynamic loads. We are also considering incorporating the 4 FSRs around the perimeter much like how StepManiaX is designed.

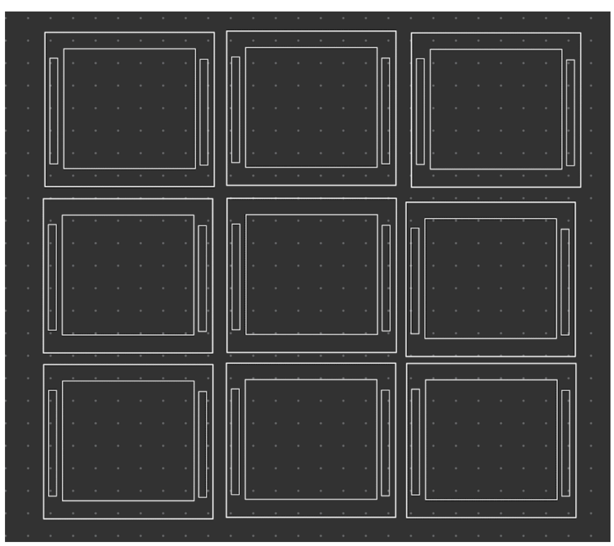


Figure 9.5 *Pairs of FSRs on opposite ends chosen integration layout*

On the microcontroller side, current integration has focused on establishing a functional prototype using an Arduino Leonardo on a breadboard. So far, we have demonstrated input registration from a single direction using one FSR. This integration is shown below. This served as a proof-of-concept to validate both the sensor circuit and the USB HID signal pathway. Although the system supports nine directional inputs in total, we have deferred full-scale integration until we finalize our PCB design. Once one direction proves stable, the same circuit can be scaled and replicated for the remaining eight inputs.

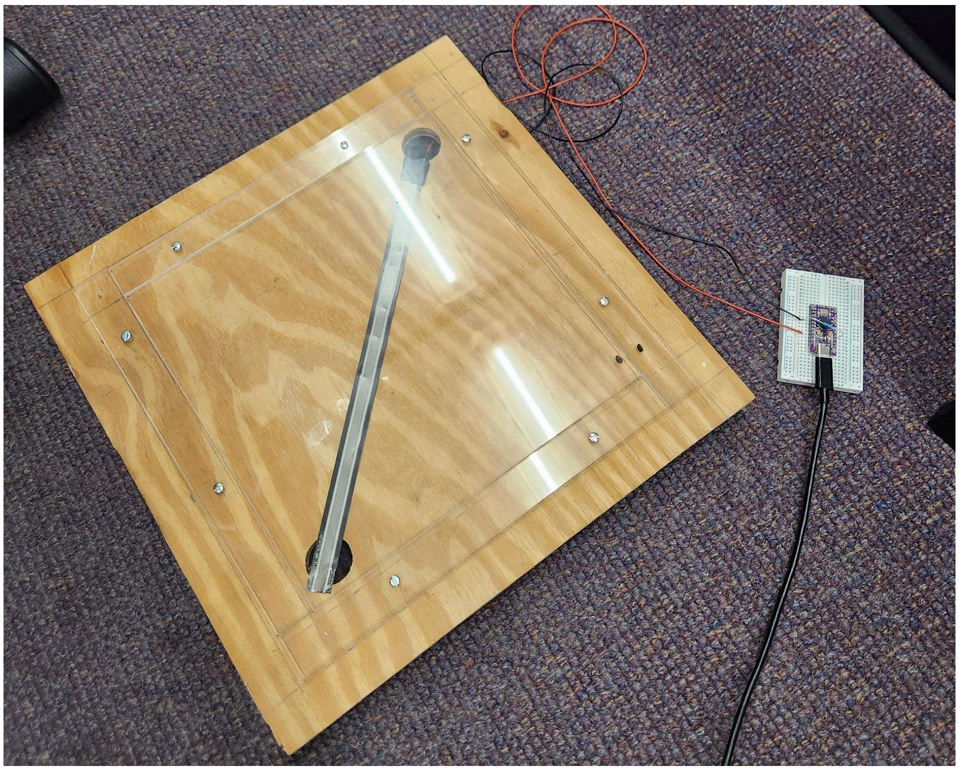


Figure 9.6 *Mini demo video pad with breadboard and single FSR integration*

In SD2, we plan to move away from breadboard prototyping toward soldered perf boards and eventually a custom PCB that will handle sensor conditioning, signal routing, and USB HID communication in a more compact and robust layout. We will also need to incorporate voltage regulation, ESD protection, and possibly signal smoothing to minimize noise and false triggering. These design considerations will be tested incrementally as we migrate from a one-sensor testbed to a fully wired 9-direction pad. Additional hardware reliability metrics such as actuation force, response time, and thermal consistency will be evaluated once all inputs are functional.

9.3.2 Software

After creating the input mapping and after the creation of the game, our first step was to conduct a unit test of having at least one arrow to light up on the screen when a certain keycode is pressed. At first the arrow would not light up but after some testing we have the arrow to light up without fail when pressing the button if there is a signal being read.

The next step was to test the arrow detection system and calibrate it correctly for the players timing. We had multiple trial and error of what threshold values to have for each of the different scores between “Perfect”, “Good”, “Bad”, “Miss”. Once we had a comfortable threshold, we set it and tested the game as a whole to make sure that it worked for any and all arrows that would appear.

In terms of the computer vision aspect we first had a computer vision model that is able to, using a variety of points, detect what type of pose is being shown on the camera. This is due to using the MediaPipe computer vision model implemented in conjunction with the camera. We then manipulated the model to our liking through a series of test cases so that it is able to detect poses for example such as head scratching. The poses we will implement are: “What?”, “Muscle Man”, “Point Up Pose”, “Samurai Pose”, “Mantis Pose”, and “Tough Guy”.

9.4 Optoelectronics Feasibility Study and Testing

The primary aim of the S.T.E.P.S system is to function efficiently as a rhythm game playable in both home and arcade environments, with a cost and power-efficient architecture. A critical aspect of this is the optical and illumination subsystems. When designing the lens, it was determined that sourcing an off-the-shelf aspherical M12 lens was the most practical and cost-efficient strategy, rather than developing a custom-manufactured optic. A custom aspherical lens would have been expensive and time-consuming to prototype, while commercial M12 aspherical options still met the project’s required field/depth of view, resolution, and IR transmission for MediaPipe performance.

Cost efficiency also drove the camera selection. Initially, higher priced Arducam modules were considered, but research found the SVPRO AR0234-based global shutter camera to provide comparable performance for a lower price. This module supports M12 lenses, has no IR-cut filter, and maintains the necessary 3µm pixel pitch and 60 FPS frame rate for reliable gameplay tracking.

For the illumination system, a key design consideration was ensuring consistent player coverage while avoiding excessive instantaneous current demands. Arcade environments can have variable or dim ambient lighting, making active illumination important for tracking. Initially, larger LED panels inspired by other rhythm games (like Dance Around) were explored. However, powering all zones simultaneously would have increased the peak current demands to around 69W,which would place higher demands on the power supply and wiring. By instead lighting only one zone at a time, the peak current was reduced to around 18W, supporting simpler and more efficient system integration.

Additional layout options were explored, including ring lights, edge-mounted LED bars, and corner-mounted strips. These were evaluated for feasibility, wiring complexity, and potential glare or reflections. Ultimately, monitor-mounted LED arrays offered the best compromise of player coverage, mechanical simplicity, and consistent geometry relative to the camera, while supporting portability.

9.4.1 Design Requirements and Specifications

The optical and illumination subsystems must work together to ensure accurate, consistent pose tracking under diverse conditions. For the optical design, the target was to achieve at least 3 pixels per millimeter resolution, a frame rate of 60 FPS minimum, and a horizontal field of view near 77-93 degrees to reliably capture the entire 2.9 m x 2.9 m tracking area surrounding the dance pad from a 1.83 m distance. The lens was expected to maintain minimal geometric distortion, with sufficient sharpness to resolve ~1mm limb features. For the illumination system, the design aimed for at least 85% brightness uniformity, 850 nm near-IR wavelength for user comfort, and an equivalent brightness of 300 lux at the player distance, while managing power with a time-multiplexed scheme to limit peak current draw. These requirements guided component selection, mechanical layout, and optical design.

9.4.2 Optical System Design

The purpose of the vision subsystem is to capture spatially accurate, high-contrast images of the player's movements during gameplay, supplying reliable data to the MediaPipe pose estimation framework for style-based scoring. The optical design is critical for minimizing ambiguity in landmark detection, reducing the computational burden on the pose estimation model, and improving real-time responsiveness.

The optical system targets a minimum of 3 pixels per millimeter to resolve features approximately 1 mm in size on the dance pad. With a pixel pitch of 3.0μm, the required image-space resolution is:

3 x 3μm = 9.0 μm

The system magnification M is then:

$$M = \frac{Image\ size}{object\ size} = \frac{9.0\mu m}{1mm} = \frac{9.0}{1000} = 0.009$$

The focal length was calculated using an equation that has the horizontal field of view in mind:

$$f = \frac{d * w_s}{W} = \frac{1830*5.67}{2900} \sim 3.63mm$$

Where:
f is the focal length
d is the camera to target distance
w_s is the width of the 1/ 2.6'' image sensor
W is the required horizontal scene width

Considering the distance from the sensor and the dance pad, the depth of field is a very important factor when choosing the right lens. Multiple steps were taken to get this value. The calculated focal length was used to determine the depth of field.
Hyperfocal distance H:

$$H = \frac{f2}{N*c} + f = \frac{(3.63)^2}{2*0.005} + 3.63 = 1321.32mm$$

Where:
f = focal length
N = aperture
c = circle of confusion for a 1/ 2.6'' sensor
Near Limit:

$$D_n = \frac{1321.32*1830}{1321.32+(1830-4)} = 768.19mm$$

Far Limit:

$$D_f = \frac{1321.32*1830}{1321.32-(1830-3.63)} =- 4787.68\ mm\ (\infty)$$

Through these calculations the depth of field was determined to be 856mm → ∞

This confirms a wide-angle lens with a focal length closer to 4 mm is needed, consistent with the prototype lens choice (CIL042). That lens, with a diagonal FOV near 97° (translating to roughly 88° horizontal on the AR0234 sensor), ensures the system can fully view the dance pad area without demanding excessive installation height or extreme tilt angles.

Despite this favorable depth of field, the limited vertical coverage poses a practical constraint. A wider angle lens, such as a 3.2 or 2.8mm focal length, may provide a better balance between full scene coverage and sufficient image resolution for reliable AI tracking. These lenses offer increased vertical field of view, ensuring the player remains fully visible anywhere on the pad while maintaining sufficient detail for pose estimation. The same DOF calculations were done with the 3.2 mm focal length as well.

$$H = \frac{f2}{N*c} + f = \frac{(3.2)^2}{2.3*0.005} + 3.2 = 893.63mm$$

Where:
f = focal length
N = aperture

c = circle of confusion for a 1/2.6’’ sensor
Near Limit:

$$D_n = \frac{893.63*1830}{893.63+(1830-3.2)} \sim 602mm$$

Far Limit:

$$D_f = \frac{893.63*1830}{893.63-(1830-3.2)} =- 1752.46\,mm\,(\infty)$$

The selected 3.2 mm M12 aspherical lens was determined to be optimal for the S.T.E.P.S system’s vision needs. The primary requirement was to achieve a wide horizontal field of view (~87°) to cover a 2.9 m x 2.9 m player tracking area from a camera placed 1.8 m away. Depth of field calculations showed that, with an aperture of f/2.3 and a circle of confusion of 0.005 mm(typical for a 1/2.6’’ sensor), the hyperfocal distance is approximately 893 mm. This means that any subject located 0.6 m or farther from the camera remains in acceptable focus, perfectly encompassing the full player’s body in the play zone. Because of this extremely large DOF, no autofocus is required, and fixed- focus imaging is sufficient for accurate pose tracking using MediaPipe. The 3.2 mm lens also provides acceptable pixel density for AI-based landmark detection and maintains real-time imaging performance at 60 FPS when paired with the global shutter AR0234 camera. Therefore, this lens was selected for its excellent balance of field coverage, focus stability, and cost efficiency.

The chosen lens (CIL034-F2.3- M12ANIR) was evaluated against the project’s optical requirements. Table 9.1 summarizes how it meets the key constraints related to the field of view, resolution, DOF, and AI tracking support.

Table 9.1 *Lens Selection Justification*

Lens Spec Requirement	3.2 mm M12 Lens Performance	Meets Requirement
Horizontal Field of View (FOV)	~ 87 ° at 1.8 m distance	Yes
Depth of Field (DOF)	~ 0.6 m to ∞ (fixed focus)	Yes
Pixel resolution	~2.5- 3 px/ mm over 2.9 m FOV	Yes
Frame Rate Compatibility	60 FPS @ 1080p with global shutter	Yes
AI Tracking Performance	Clear full-body pose detection across tracking zone	Yes

Aberration/ distortion Control	Minor edge distortion, acceptable for vision-based tracking	Mostly
Cost and availability	~ $40	Yes

***Table 9.2** Depth of Field Calculation for 3.2 mm f/2.3 lens at 1830 mm subject distance.*

Parameter	Value	Units
Focal Length	3.2	mm
Aperture	2.3	—
Circle of Confusion	0.005	mm
Subject Distance	1830	mm
Hyperfocal Distance	893.63	mm
Near Focus Limit	601.14	mm
Far Focus Limit	∞	mm
Effective DOF	0.6 m to ∞	meters

The calculations in table 9.2 confirm that the fixed-focus lens design is sufficient for full-body imaging at the target distance, eliminating the need for autofocus and ensuring reliable MediaPipe performance.

9.4.3 Illumination System design

The optical system's effectiveness in detecting pose landmarks is significantly influenced by the quality and consistency of scene illumination. To ensure accurate and robust tracking performance in a variety of ambient lighting conditions, the S.T.E.P.S system employs a custom-designed near-infrared(NIR) LED illumination array operating at 850 nm. This wavelength was chosen because it is largely invisible to the human eye, minimizing player distraction, while maintaining high sensitivity on the AR0234 global shutter RGB camera used in the vision subsystem.

The illumination system consists of four time-multiplexed IR LED zones, positioned along the top, bottom, left, and right edges of the display frame. Each zone is independently controlled and provides directional illumination across the play area. This setup enables zone-based pose segmentation by sequentially lighting one direction at a time in sync with the camera capture, without requiring a separate LED per pad tile.

These zones are time-multiplexed using a custom microcontroller driver circuit to avoid interference between overlapping beams. Each zone activates sequentially at a high enough frequency(60-90Hz) ensuring the player remains well-lit from multiple directions while minimizing flicker and thermal buildup. This time multiplexed design avoids overexposure from simultaneous LED output, improves lighting uniformity, and enhances MediaPipe's ability to consistently track pose landmarks across the entire play area.

Each LED zone is mounted around the perimeter of the game area, positioned to illuminate the player from multiple angles. While final mounting angles are still being refined, the design assumes slight inward tilts to maximize coverage uniformity. While the total LED power draw is moderate, concentrated NIR output can still result in localized heating. To mitigate any user discomfort, such as excessive sweating due to proximity to strong IR emitters, the system spreads its output over time and space. Power measurements and eye safety limits are being reviewed against IEC 62471 photobiological safety standards. Preliminary analysis indicates the system remains well below Class 1 limits, but formal verification will be completed in SD2 through irradiance measurements and angular spread modeling.

9.4.4 Optical Illumination System Testing and Verification

The optical and illumination system will be carefully tested to ensure they meet these engineering requirements before integration into the full S.T.E.P.S cabinet. Testing will focus on verifying that the field of view fully covers the 2.9 m x 2.9 m tracking area surrounding the dance pad, confirming spatial resolution supports ~ 1mm feature detection, measuring illumination uniformity across the player's body, ensuring time-multiplexed zones maintain seamless perceived lighting without flicker, and verifying that a minimum of 300 lux equivalent illumination is achieved at a player distance of 1.83 m.

Geometric and image quality verification will be carried out using a printed calibration checkerboard pattern to check the camera's coverage, measure geometric distortion, and confirm the intended horizontal field of view. Separately, a uniform reflectivity target will be used to verify even illumination across the dance pad area. Pixel intensity measurements will then be analyzed frame-by-frame to confirm both brightness uniformity and signal-to-noise performance, ensuring no major shadows or hotspots occur during typical player movements. These tests will be performed on individual components during senior design I, then the integrated system for senior design II.

9.4.5 Qualitative Distortion Assessment

To evaluate optical distortion introduced by each lens, checkerboard calibration targets were imaged at the standard system mounting height (~63 inches) using the 3.2 mm, 2.8 mm, and 4.2 mm lenses. While all three lenses produced sharp central focus under proper lighting, noticeable differences in geometric fidelity were observed.

The 2.8 mm lens, with its 120° horizontal field of view, exhibited significant barrel distortion, particularly near the image periphery. Straight lines on the checkerboard became visibly curved, and square tiles appeared stretched at the corners. This level of distortion is known to interfere with AI-based pose estimation, especially in landmark tracking for limbs and extremities near the edge of the frame.

By comparison, the 3.2 mm lens showed milder distortion, though some barrel warping was still evident. A checkerboard test image captured with this lens demonstrated minor stretching of squares near the frame edges, but central geometry remained well-preserved. This suggests that while the 3.2 mm lens is not perfectly rectilinear, its distortion is within acceptable limits for MediaPipe's pose estimation pipeline and does not require software-based undistortion for prototype use.

The 4.2 mm lens offered the least distortion, but its narrower field of view risked cutting off portions of the dance pad. As such, the 3.2 mm lens was selected as an optimal balance between geometric fidelity and scene coverage. If future development requires full-scene imaging with wider-angle lenses, OpenCV-based undistortion tools may be explored for real-time rectification.

9.4.6 IR Sensitivity Verification

To validate that the chosen camera (AR0234 NoIR configuration) could detect infrared (IR) light from the 850 nm LED illumination system, a simple IR sensitivity test was performed. A known IR-emitting source (a standard television remote and active LED strip segment) was placed in close proximity to the lens under typical room lighting. When observed through the camera's live feed, both sources appeared visibly illuminated on the image sensor confirming that 850 nm wavelengths were reaching the sensor and not being blocked by an internal IR-cut filter.

This observation confirms the optical subsystem's compatibility with the NIR illumination zones. Final irradiance and synchronization validation will be performed in SD2, but the preliminary results affirm that the system can reliably image in the near-IR spectrum and support pose tracking under IR-only conditions.

9.5 Overall Integration

9.5.1 Hardware

The hardware integration of the STEPS system focuses on the seamless coordination between the Pad PCB, FSR input mechanisms, RGB LED feedback, display, and power delivery. The Pad PCB serves as the central hub for all peripheral connections, routing input signals from the FSR sensors and output signals to the RGB LED strips embedded in the pad. The camera module and its custom lens are mounted in alignment with the display, while the LED strip around the display receives simple on/off signals from the MCU through the LED PCB. A central 12V, 5A SMPS provides the primary power supply, safely distributing energy to both high-current loads like the LED strips and low-power components like the microcontroller. By centralizing both control and power at the Pad PCB, the overall design ensures minimal latency, reliable sensor reading, and predictable system performance under continuous use.

9.5.2 Optical and Illumination Systems

The optical and illumination systems will be integrated with the broader S.T.E.P.S architecture through synchronized control managed by the system microcontroller. The existing time-multiplexed illumination strategy, previously described, will be coordinated with camera exposure to maintain uniform, flicker-free NIR lighting. Mechanically, the camera module and LED arrays will be mounted to the monitor structure to ensure consistent geometry and coverage across all play environments. This integration supports real-time pose estimation with minimal latency, allowing seamless interaction between hardware and gameplay logic.

9.5.3 Software

After ultimately deciding on using the MediaPipe interaction of computer vision, the camera will be able to send video footage to the computer which will then be interpreted by the computer vision model for pose detection. What will happen behind the scenes is that by using the camera that is pointed towards the player playing, the camera will capture live video footage of the person to which afterwards will be sent to the computer that will be running the MediaPipe computer vision model that we use for pose detection. How the computer vision model works is that by having a series of points or landmarks being applied to the person though the computer vision model itself, then using some code, those points or landmarks will then be interpreted as poses depending on the threshold set.

Afterwards, inside the game itself we will have a game implementation of the pose recognition system by having indicators on screen that tell players when to perform the certain pose that appears. Depending on the timing of the pose being performed, then an accuracy score between “Perfect”, “Great”, “Good” or a “Miss”.

For the game integration itself, we will be using Godot and we will also be using Git for our version control. While creating the game we created a series of nodes and sprites that will start from the bottom of the screen and work their way up in a scrolling motion.

As these notes are scrolling upwards, there is a detection system for deciding the timing that players who play the game decide the score to give them depending on the timing threshold. Whether that be “Perfect”, “Great”, “Good” or a “Miss”.

Furthermore, we added a system that allows for the creation of songs. By having the inputs being pressed, the code will allow the user to add the arrow at that specific timing. Although this system is not fully implemented yet with the UI, we plan on doing so during SD2 but we do however have the script done for it.

Overall our future plans for SD2 will be more focused on the implementation of song creation as well as the creation of a better main menu than we have now.

9.6 Plan for SD2

9.6.1 Hardware

Weeks 1–3

Hardware development will begin with the fabrication and bring-up of the main Pad PCB. During week 1, we plan to finalize and submit the PCB order to manufacturers such as JLCPCB, allowing time for fabrication and shipping. Once received, the focus will be on assembling the board and verifying basic electrical functionality. Key bring-up tasks include checking voltage regulation, USB HID enumeration, and sensor input signal integrity.

By week 3, if any issues arise with the initial PCB revision, we will allocate time to make necessary corrections and resubmit an updated board. In parallel, we will begin early mechanical work, including cutting sensor foam, aligning top acrylic panels, and preparing mounting layers for the FSR sensors.

Weeks 4–6

Once the PCB is validated, we will shift focus to full integration of the FSR sensors and the RGB LED feedback system. Directional pad inputs will be tested end-to-end, verifying whether foot pressure on panels consistently triggers USB inputs on the PC. Around week 5, we will begin wiring and securing the LED system, ensuring zoned control between center pad LEDs and the outer illumination strips.

Power delivery will also be validated during this phase. We will monitor thermal behavior, voltage stability, and current draw under load to ensure safe and consistent operation of the system when all lights and sensors are active.

Weeks 7–10

These weeks will be focused on system-wide integration between the input system, LED feedback, and game software. The RGB LED zones will be programmed to reflect gameplay events (e.g., scoring feedback, pose prompts), and stress tests will be conducted to measure timing accuracy, communication reliability, and input responsiveness during gameplay.

Cable routing, connector strain relief, and physical durability checks will also be performed to verify that the full system can withstand repeated foot strikes and extended play sessions without mechanical failure.

Weeks 11–12

Final hardware debugging, mechanical reinforcement, and gameplay validation will be conducted during this period. Key deliverables include ensuring all FSR panels are mechanically aligned and electrically reliable, verifying that LED feedback occurs in sync with in-game events, and confirming that the power system remains stable across sessions. Additionally, we will run full-system QA tests to check for loose components, exposed wiring, and proper enclosure sealing.

9.6.2 Optical and Illumination Subsystems

Weeks 1–3

The first three weeks will focus on camera setup and initial calibration. The camera will be physically mounted above the 2.9 m × 2.9 m play area at approximately 1.8 m height. Mount alignment, angle tuning, and mechanical fastening will be completed to ensure stable and complete field-of-view coverage.

Following installation, field-of-view calibration using printed checkerboard targets will be conducted. This step will validate spatial resolution and ensure that the camera meets the ~1 mm feature detection accuracy needed for effective pose estimation using MediaPipe.

Weeks 4–5

Once the camera is calibrated, we will install and configure the 850 nm IR LED illumination zones. These zones will be arranged to provide even coverage of the entire play area, based on CAD-modeled light cone distributions. Physical alignment and beam spread will be verified against these models, and power measurements will be taken to confirm irradiance levels remain within safety thresholds.

After installation, we will test for interference between IR illumination and pose estimation accuracy under various ambient lighting conditions. If necessary, brightness and duty cycles will be adjusted using zoned control and time-multiplexing strategies.

Weeks 6–8

Integration with the software pipeline begins in this phase. Real-time video from the camera will be processed using MediaPipe, and body keypoints will be passed to the gameplay logic for triggering pose-based events. We will validate synchronization between pose inputs and the game system, as well as evaluate any camera lag or frame drops.

Metrics such as zone irradiance uniformity, tracking consistency, and image clarity will be quantitatively evaluated using calibration tools and comparison charts.

Weeks 9–12

Final adjustments, validation, and debugging will be performed. This includes checking for optical distortions, verifying photobiological safety of the IR LEDs, and stress testing MediaPipe under rapid player movement. Any remaining issues related to tracking loss, misclassification, or latency will be resolved before the system is declared ready for demonstration and integration with gameplay scoring logic.

9.6.3 Gameplay system

Week 1-3

During SD2 with respect to the game integration aspect, the first three weeks will consist of improving the game integration on the coding side. We will be working on improving the arrow key integration, improving on adding a series of arrows that will appear from the bottom of the screen and move their way up, as well as improving system in the game that is able to detect the arrows that need to be clicked and if they are clicked depending on the timing they will receive a certain amount of points.

In terms of arrow integration, we will be working on where the various arrows will be placed on the screen, making the arrows prominent and large enough for the users to be able to see them without having difficulty as to determining what arrow they need to be pressing while playing the game.

In terms of the series of arrows appearing from the bottom of the screen and making their way up, we have plans using various trial and error methods to be able to complete this task. At the end of the trial and error process the goal is to finalize the implementation of the detection arrow key system.

Weeks 4-6

Weeks four through six will consist of having a completed and finalized arrow key integration, after some trial and error have a plan of integrating both the series of arrows and the arrow detection system.

We will accomplish this by having just a few arrows first as a test run and work on the detection system until it is able to pass the test case. Once the detection system is able to pass the unit test, the next step will be to complete the series of arrows scrolling across the screen.

Weeks 7-10

In the following weeks seven through ten, we will be focusing on implementing all of the pose events into the game. There are a total of 6 different poses that we plan on implementing for the pose detection system. Those poses include: “What?”, “Muscle Man”, “Point Up Pose”, “Samurai Pose”, “Mantis Pose”, and “Tough Guy”.

The goal is to have the poses show up on screen when the event happens using an event handler mechanism implementation. Furthermore have an accuracy implementation sequence that depending on the timing of performing the pose adds the points scored to the players total score towards the end of the song being played.

Weeks 11-12

These weeks we will be doing extensive testing on all 6 of the different poses on how accurate they are as well as activation during gameplay. Furthermore, we will be doing more testing with the arrows and the pose events.

How we will accomplish this, is by doing a series of unit tests making sure that each action is able to pass a test that we assign it. Then after it has passed one, we test for edge cases so that they are accounted for.

Week 13

In this week, we will be focusing on making full charts with different difficulties, with each song having different difficulties and charts for each difficulty.

Like in our software diagram, ***Figure 2.8***, we will work on creating a menu that will allow the player to be able to choose a difficulty for songs. The different difficulties that will be included in the menu will be easy, medium, or hard. The different difficulties will change the game based on the amount of arrows being created. Easy will have a small number of arrows appear at any given time, medium will have a moderate increase to the number of arrows, and hard will have the most increase compared to easy.

Weeks 14-16

In the final weeks we will be asking different people whether that be our advisors, friends and family, and even test the game on our own for play testing and quality assurance testing. After this, depending on the feedback received, we can change and modify as need be.

10. Administrative Content

10.1 Budget

We are aiming to limit the budget of this project to $500. *Table 10.2* as shown below lists the bill of materials. Although we want to minimize costs, we also want to ensure we have materials that have a good enough quality so that they are reliable and efficient. This includes the framework (hard materials), PCB, RGB LEDs, LED ring, camera module, and force-sensing resistors. If any of these were bought very cheap without taking into account better affordable options, we would face consequences.

10.2 Bill of Materials

Table 10.1 *Overall Project Itemized Bill of Materials*

Item	Dimensions	Estimated Unit Cost	Quantity	Estimated Total Cost
SVP AR0234 Sensor	38mm x 38mm	$65.99	1	$65.99
Commonlands CIL034 lens	20mm x 14mm	$47.00	1	$47.00
SMD3528 Near Infrared LEDs	5m x 0.01m	$45.99	1	$45.99
PCB	undetermined	$45.85	1	$45.85
Force-Sensing Resistors	12.7mm x 57mm	$5	36	$90
Plywood	36 `` x 38 x ¼ ``	$50	1	$50
Aluminum Square tubing	38`` x 38`` x 1/8``	$10	6	$60
Polycarbonate Sheets	11.75 ``x 11.75`` x 0.25 ``	$0	9	$0(already have)
Non-Slip Rubber Mat	2ft x 4ft x 3mm	$20	1	$20

Table 10.2 *Master Controller Board (1 required) Itemized Bill of Materials*

Reference Designator	Quantity	Component	Estimated Cost (USD)
U1	1	ATmega32U4RC-AU Microcontroller	$4.50
Y1	1	16 MHz Crystal	$0.40
Subtotal			**$5.90**

Table 10.3 *Power Hub Board (1 required) Itemized Bill of Materials*

Reference Designator	Quantity	Component	Estimated Cost (USD)
U1	1	MP24894 LED Driver IC	$2.00
U2	1	LM2576T-5.0 5V Regulator	$1.50
Subtotal			**$3.50**

Table 10.4 *Tile Board (9 required) Itemized Bill of Materials*

Reference Designator	Qty per Board	Total Qty	Component	Est. Cost per Unit	Total Est. Cost (USD)
U4	1	9	ATtiny85-20PU Microcontroller	$1.80	$16.20
D6-D14	9	81	APA-106-F5 Addressable LED	$0.25	$20.25
Subtotal					**$36.45**

Table 10.5 *PCB board total cost evaluation table*

Board	Estimated Cost (USD)
Master Controller	$5.90
Power Hub	$3.50
All 9 Tile Boards	$36.45
Grand Total (Electronic Components)	**$45.85**

10.3 Distribution of Worktable

Table 10.6 *Distribution of responsibilities of each member for this project*

Name	Major	Responsibilities
Andres Abrams	Computer Engineer	Software Assistant
		Software Design and Implementation
		Game Design

Name	Major	Responsibilities
Blake Whitaker	Electrical Engineer	Hardware Lead
		PSU Design and Implementation
		PCB Design

Name	Major	Responsibilities
Jani Jon Lumibao	Computer Engineer	Hardware Assistant
		MCU Selection and Implementation
		Embedded Programming

Name	Major	Responsibilities
Christopher Solanila	Computer Engineer	Project/Software Lead
		Website Design and Management
		Software Design and Implementation
		Game Design
		Computer Vision Implementation

Name	Major	Responsibilities
Kaila Peeples	Photonics Engineer	Lens Design and Simulation
		Camera Module Integration
		Optical and Illumination System Optimization
		Image Quality Calibration
		Illumination planning and testing

10.4 Project Milestones for SD1 and SD2

10.4.1 Project Milestones for SD1

Table 10.7 *Project Milestones SD1*

Due Week	Advancement
1	Group Creation and have base idea
2	Researching and Innovating based off idea
3	Divide and Conquer Document completed, Have at least 1 committee member
4	Meet for revisions (if any) of Divide and Conquer Document with committee members, upload revised document into group website
4-5	Individual Research
6	40 pages finished

7	Meet for revisions (if any) with committee members
7-8	Testing of components
9	100 pages finished turn in Midterm Report
10	Meet for revisions (if any) with committee members
10-11	Start on video, finishing touches to 120 page document
12	150 page document finished, mini video finalized

10.4.2 Project Milestones for SD2

Table 10.8 *Project Milestones for SD2*

Target Completion Week	Advancement
2	Camera system & PAD Completion
4	Camera Hardware Completion
5	NIR illumination System Completed
6	Game Completion, Optical Calibration, and Camera detection
7	Pad and Game Integration
8	Camera-to-Game Integration
8	NIR illumination System Integration
12	Full System Testing and Debugging
16	Final Day and Live Demo

10.4.3 Development Roadmap

We aim to complete the following specific tasks over the course of SD1 and early SD2. These objectives are listed in the rough order we plan to execute them, starting from component acquisition and hardware assembly to software integration and vision-based features.

1. **Acquire all essential components:** Order FSR sensors, RGB LEDs, MCU, USB connectors, power regulators, 850nm LED strips, and a suitable camera module according to the finalized bill of materials.

2. **Design and fabricate the dance pad platform:** Construct the pad frame using durable plywood and aluminum supports. Mount the top transparent acrylic panels and add non-slip material underneath to ensure player stability.

3. **Design a custom PCB schematic:** Use KiCad or similar EDA tools to create a schematic and board layout connecting all pad components. Allocate GPIOs for sensors and LEDs, and ensure proper voltage regulation and USB connectivity.

4. **Order and assemble PCB:** Send the PCB design to manufacturers such as JLCPCB for fabrication. Once received, solder all required components and headers for interfacing with sensors, LEDs, and the MCU.

5. **Write and test microcontroller firmware:** Develop firmware capable of reading analog signals from FSRs and converting them into digital inputs. Implement USB HID functionality to send these signals as keypress events to the PC.

6. **Test PCB + FSR response time:** Run debugging scripts to verify that each sensor input registers with low latency (<10ms), ensuring responsiveness suitable for rhythm gameplay.

7. **Integrate pad input with PC:** Confirm successful USB HID functionality by testing real-time inputs using diagnostic tools or key input test pages on a computer.

8. **Initialize the GitHub repository:** Set up a shared repository with version control for the team. Include directories for firmware, hardware schematics, and game engine development.

9. **Begin a Godot project for the rhythm game engine:** Set up a new project in Godot 4.3 with basic scene structure, a title screen, placeholder assets, and a framework for spawning notes in sync with music.

10. **Map keyboard inputs to note triggers:** Bind USB HID inputs to game actions in Godot. Simulate gameplay using a keyboard before full pad integration to test

timing and hitbox accuracy.

11. **Design and implement a basic UI:** Build out essential game UI elements such as a song selection screen, live scoring display, and feedback for note accuracy using visual and textual cues.

12. **Write a chart parser and loader:** Develop functionality to import step charts from external files (e.g., JSON format) and dynamically spawn notes according to their timing and track.

13. **Connect the pad to the game:** Integrate the full dance pad input with the game logic, ensuring each physical press correctly triggers corresponding note hits and feedback during gameplay.

14. **Integrate camera with Raspberry Pi:** Connect the camera to the Raspberry Pi and confirm live video can be accessed and streamed to the PC via a local network or direct USB interface.

15. **Use a pre-trained pose detection library:** Utilize computer vision tools such as OpenCV, MediaPipe, or OpenPose to extract body key points from the live camera feed in real time.

16. **Detect key poses and gestures:** Program the system to recognize predefined poses like the Muscle Man, What?, Mantis, and Samurai poses based on body key point configurations.

17. **Develop a basic Style Score system:** Create a scoring extension that awards bonus points for expressive poses and movement patterns detected during gameplay.

18. **Build a simple chart editor UI:** Design a chart editing tool that allows users to import songs, manually place step notes, and export them into a compatible format for game testing.

19. **Design a custom aspherical lens system:** Develop a lens assembly capable of resolving fine motion (~1mm accuracy) across a 40cm field of view when placed 1.8 meters away.

20. **Generate accurate ray layouts in Zemax:** Simulate optical performance of the lens system in Zemax to evaluate sharpness, chromatic aberration, and field distortion.

21. **Select a low-cost camera sensor:** Choose a cost-effective camera module and tune its settings to reduce resolution and field of view to ≤1.5MP, optimizing frame rate and bandwidth.

22. **Prototype and test multiple lens configurations:** Assemble and test several lens-camera combinations to confirm alignment and capture quality are acceptable for pose detection.

23. **Design a time-multiplexed LED illumination system with zoned control:** Build a lighting setup that evenly illuminates the player during movement while avoiding excessive glare or power draw.

24. **Quantitatively evaluate optical performance:** Use standardized test charts and calibration tools to assess image clarity, distortion, and pose detection reliability in the intended environment.

25. **Document each milestone:** Maintain clear documentation through GitHub commits, photos, and regular updates, ensuring the project remains transparent and demo-ready for SD1 and SD2 reviews.

11. Conclusion

The Style Tracking Expressive Pad System (STEPS) project was envisioned as a modern, expressive, and feature-rich reimagining of the arcade dance pad. Rooted in a deep appreciation for rhythm-based games and performance-driven technology, STEPS seeks to push the boundaries of what dance gaming can be, not just in terms of challenge and skill, but also in terms of artistic expression, immersion, and interaction. Unlike standard arcade systems that limit players to four or five directional panels, STEPS introduces nine directional inputs, expanding gameplay into a new physical dimension that encourages creativity, agility, and style. This redesign opens the door to both casual users who want to explore dance playfully and competitive users aiming to master complex movement patterns.

Inspiration for STEPS came from analyzing a wide range of existing arcade machines. Some machines offer RGB lighting effects. Others offer unique gameplay variations like double panels or hand sensors. However, there was a lack of unified systems that could combine style-based expression, advanced lighting, enhanced input granularity, and computer vision.

With STEPS, we aim to bring all of those technologies together into a cohesive and upgradable platform. The result is a product that is not just a dance controller, but a performance stage, an interactive installation, and a learning tool all in one.

The system includes nine FSR-based directional input zones arranged ergonomically to support full-body footwork. The addition of force-sensitive resistors instead of traditional microswitch panels provides more precise readings, enabling us to capture not just when a player steps, but also how hard they step.

This opens up new design possibilities for rhythm games that can respond dynamically to the energy or weight of a dancer’s movement. Moreover, each direction includes paired FSRs, further enhancing accuracy and supporting redundancy or calibration features down the line.

As the MCU of our system, we use a single Arduino Leonardo microcontroller, chosen for its native USB HID support and simple integration with PC-based rhythm games. Unlike serial communication or custom drivers, USB HID allows the STEPS system to be recognized as a standard input device on most operating systems, which simplifies the development and testing process. Future upgrades could include Bluetooth LE or Wi-Fi for wireless play, making setup more flexible and less cluttered.

The STEPS system is powered by a 12V, 5A external enclosed Switch-Mode Power Supply (SMPS), chosen for its efficiency, safety, and ability to handle the system's combined high-current and low-current demands. It provides stable power for both the RGB LED strips and sensitive components like the Arduino Leonardo. Proper filtering and decoupling built into the custom PCBs ensure noise from the SMPS does not affect performance, making this a reliable and safe power solution for the dance pad.

The use of RGB LED strips around the dance pad and around the display unit further boosts immersion. While the LED panels currently support basic on/off functionality for illuminating the player, their modular design allows us in the future to program it to have advanced lighting effects and animations. These lighting elements also enhance computer vision performance by providing backlighting and visual contrast for foot tracking.

A camera module with a custom-made lens faces the pad and helps track limb movement, enabling gesture recognition or footwork quality analysis and maximizing the camera’s accuracy for computer vision, creating a bridge between motion capture and rhythm-based play.

By separating the input detection and visual feedback functions, the STEPS architecture maintains modularity and avoids overloading the microcontroller. The Pad’s PCBs focus solely on reading FSR data and controlling basic LED states, while future improvements may offload animations to a dedicated LED controller. This separation ensures cleaner timing and responsiveness, especially in latency-sensitive rhythm games.

From a user experience standpoint, STEPS is designed to feel intuitive and engaging. New players are given freedom to explore a wider movement space without sacrificing accuracy, and experienced players can use subtle foot pressures or directional changes to add flair to their performances. Beyond entertainment, STEPS has educational potential, whether in dance studios for training foot placement, or in physical therapy settings for improving balance and movement coordination. Its flexibility makes it a useful tool for both game designers and movement educators.

Another benefit of STEPS lies in accessibility. By integrating pressure-sensitive input rather than forceful switches, the pad requires less physical impact, making it more comfortable for a wide range of players. With further refinement, the system could incorporate audio feedback, accessibility overlays, or adjustable lighting profiles to support neurodivergent users or players with visual impairments.

Looking forward, there are many directions this system can evolve. For instance, the addition of haptic feedback (e.g. vibration) could give players tactile response for combo streaks or rhythm timing.

On the software side, pairing STEPS with AI dance recognition models could lead to training tools that give real-time tips or scoring based on style rather than accuracy alone. More advanced LED animations, sound-reactive lighting, or player-generated choreography libraries could make STEPS a platform for dance-based expression in gaming, performing arts, or interactive museum installations.

In conclusion, the STEPS dance system is more than just an upgrade, it’s a forward-thinking redesign of what dance input hardware can be. It introduces technical innovation without forgetting its core purpose: to make dance more expressive, more interactive, and more fun. Through carefully selected components, modular design, and a focus on player experience, we believe STEPS lays the groundwork for a new generation of rhythm-based interfaces. Whether it’s in the arcade, at a competition, in the studio, or at home, STEPS has the potential to move not just players, but the entire dance gaming community, forward.

Appendices

Appendix A - references

[1]Guerra-Filho, G. B. (2005). Optical Motion Capture: Theory and Implementation. *Revista de Informática Teórica e Aplicada (RITA)*, 12(2), 1–18.

https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=07147486b65d12c4326ccb3ad54ca612b52e1ac3

[2] *Dance around information*. RemyWiki. (n.d.). https://remywiki.com/DANCE_aROUND_Information

[3] *Dance around (AC) - bemani games - music game forums*. ZIv. (n.d.). https://zenius-i-vanisher.com/v5.2/thread?threadid=11041&page=2#:~:text=The%20games%20body%20tracking%20is,games%20that%20cater%20to%20everyone.

[4] *Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh. OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields IEEE TPAMI, 2019. https://doi.org/10.1109/TPAMI.2019.292925*

[5] *Google MediaPipe. (2024). Pose Estimation. Retrieved from = https://developers.google.com/mediapipe/solutions/vision/pose*

[6] *Step Revolution. (2024). StepManiaX. Kyle Ward Retrieved from https://stepmaniax.com*

[7] Sciotex. (n.d.). *Light colors used in machine vision*. Retrieved from https://sciotex.com/light-colors-used-in-machine-vision/

[8] Ultralytics. (2025). *A guide to camera calibration for computer vision in 2025*. Retrieved from https://www.ultralytics.com/blog/a-guide-to-camera-calibration-for-computer-vision-in-2025#:~:text=Oftentimes%2C%20they%20end%20up%20capturing,for%20real%2Dworld%20AI%20applications

[9] Flexfire LEDs. (n.d.). *Color rendering index (CRI) and LED lighting: What is CRI?* Retrieved from https://www.flexfireleds.com/color-rendering-index-cri-and-led-lighting-what-is-cri/#:~:text=CRIs%20under%2080%20are%20generally,one%20measurement%20for%20light%20quality

[10] MassedCompute. (n.d.). *How does the size of an image affect the training time of a CNN?* Retrieved from https://massedcompute.com/faq-answers/?question=How%20does%20the%20size%20of%20an%20image%20affect%20the%20training%20time%20of%20a%20CNN

[11] Supertek Module. (n.d.). *Monochrome camera vs. color: Which is best?* Retrieved from https://www.supertekmodule.com/monochrome-camera-vs-color/#:~:text=Monochrome%20cameras%20are%20often%20used,color%20cameras%20are%20better%20suited

[12] Commonlands. (n.d.). *Wide angle 3mm M12 lens CIL329*. Commonlands. Retrieved July 10, 2025, from https://commonlands.com/products/wide-angle-3mm-m12-lens-cil329

[13] International Organization for Standardization. (2010). *ISO 9241-210: Ergonomics of human-system interaction*. Retrieved from https://cdn.standards.iteh.ai/samples/52075/c30c5ea5097843ecb89a9d417f9cdab1/ISO-9241-210-2010.pdf

[14] Smart Vision Lights. (n.d.). *IEC/EN 62471 summary: Photobiological safety of lamps and lamp systems*. Retrieved from https://smartvisionlights.com/wp-content/uploads/IEC_62471_summary.pdf

[15] IEC. (2006). *IEC 60598-1: Luminaires - Part 1: General requirements and tests*. Retrieved from https://www.lisungroup.com/wp-content/uploads/2019/07/IEC60598-1-2003-A1-2006-Standard-Free-Download.pdf

[16]] Sobel, I., & Feldman, G. (1968). *An Isotropic 3×3 Image Gradient Operator*. Introduced the Sobel operator for edge detection. Retrieved from https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator

[17] Canny, J. F. (1986). *A Computational Approach to Edge Detection*. Proposed the multi-stage Canny detector with Gaussian, non-max suppression, and hysteresis. Retrieved from https://www.researchgate.net/publication/224377985_A_Computational_Approach_To_Edge_Detection

[18] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks* (AlexNet) Retrieved from https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/

[19] Hidalgo, G., et al. (2019). *Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields* Retrieved from https://www.ri.cmu.edu/publications/openpose-whole-body-pose-estimation/

[20] Google Research. (2020). *BlazePose: On-device, Real-time Body Pose Tracking Retrieved from https://research.google/blog/on-device-real-time-body-pose-tracking-with-mediapipe-blazepose/*

[21] Google AI. *MediaPipe Pose Landmarker* lightweight, real‑time 33‑landmark body pose detection. Retrieved from https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker

[22] Canva. (2025). Canva [Web application]. Retrieved from https://www.canva.com/

[23] OpenAI. (2023). ChatGPT (Feb 13 version) [Large language model].

[24] Microsoft. (2025). copilot [AI assistant]. Microsoft Corporation.

[25] Gemini. (2025, July 6). [Response to a query about rhythm game development]

[26] DeepSeek (2024), [large language model developed by DeepSeek AI].

[27] Unity Technologies. (2025). Unity (6000.0.52f1 LTS) [Computer software]. Retrieved from https://unity.com/download

[28] Godot Engine. (2025). Godot Engine (4.4.1) [Computer software]. Retrieved from https://godotengine.org/

[29] Image Engineering. (2015). *Camera tests: White Paper 1.0*. Kerpen, Germany: Image Engineering GmbH & Co. Retrieved from https://www.image-engineering.de/content/library/white_paper/cameratests_whitepaper_1.0.pdf

[30] IPC International, Inc., IPC-2221A: Generic Standard on Printed Board Design, May 2003.

[31] C. F. Coombs, Printed Circuits Handbook, 6th ed., New York, NY, USA: McGraw-Hill, 2008.

[32] Sierra Circuits, Inc., "IPC-2221 Standards in PCB Design," [Online]. Available: https://www.protoexpress.com/blog/ipc-2221-standards-in-pcb-design/.

[33] USB Implementers Forum, Device Class Definition for Human Interface Devices (HID), Version 1.11, Jun. 27, 2001. [Online]. Available: https://www.usb.org/hid

[34] USB Implementers Forum, Universal Serial Bus Specification, Revision 2.0, Apr. 27, 2000.

[35] U.S. Code of Federal Regulations, Title 47, Part 15, *Radio Frequency Devices*, Federal Communications Commission, 2023. [Online]. Available: https://www.ecfr.gov/current/title-47/chapter-I/subchapter-A/part-15

[36] Federal Communications Commission, *OET Bulletin No. 62: Understanding the FCC Regulations for Low-Power, Non-Licensed Transmitters*, Oct. 1993. [Online]. Available:

https://transition.fcc.gov/Bureaus/Engineering_Technology/Documents/bulletins/oet62/oet62rev.pdf

[37] UL Standards, *UL 62368-1: Audio/Video, Information and Communication Technology Equipment – Part 1: Safety Requirements*, Northbrook, IL, USA: Underwriters Laboratories, 2019. [Online]. Available: https://www.ul.com

[38] UL Standards, *Hazard-Based Safety Engineering (HBSE) & UL 62368-1*, [Online]. Available: https://www.ul.com/news/hazard-based-safety-engineering-hbse-and-ul-62368-1

[39] European Commission, *Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment (RoHS) Directive 2015/863/EU*, Brussels, Belgium: European Union, 2015. [Online]. Available: https://environment.ec.europa.eu/topics/waste-and-recycling/rohs-directive_en

[40] Mouser Electronics, *RoHS Guide: Understanding Restriction of Hazardous Substances Compliance*, 2024. [Online]. Available: https://www.mouser.com/rohs/

[41] IPC International, Inc., *IPC-A-610H: Acceptability of Electronic Assemblies*, Bannockburn, IL, USA: IPC, 2020.

[42] EMSG, Inc., *Understanding IPC-A-610: An Overview of the Electronics Assembly Standard*, 2023. [Online]. Available: https://www.emsginc.com/blog/understanding-ipc-a-610/

Appendix B - ChatGPT prompts and outcomes

[43] What would be the best way to create a rhythm game for a beginner? Would using a game engine be best or are there other alternatives to complete this objective?

ChatGPT said:

Creating a rhythm game as a beginner is totally achievable and a great project to learn programming, game logic, and audio synchronization. Here's a breakdown of the best approach:

🎮 Best Way: Use a Game Engine

Using a game engine is *strongly recommended* for beginners because it handles many hard parts (rendering, audio, input, timing) so you can focus on the gameplay.

Top Beginner-Friendly Game Engines:

1. Unity (with C#)

- ✅ Strong community, lots of tutorials
- ✅ Built-in support for music syncing, 2D and 3D
- ✅ Asset Store has ready-made rhythm tools
- 📘 Tutorial: *"How to make a rhythm game in Unity"* (e.g., Brackeys' or Code Monkey's guides)

2. Godot (with GDScript or C#)

- ✅ Lightweight, open-source, no royalties
- ✅ Very beginner-friendly scripting
- ✅ Good 2D support
- 📘 Search: "Godot rhythm game tutorial"

3. Scratch (for absolute beginners or younger users)

- ✅ Drag-and-drop programming
- ✅ Great for learning the logic of timing and input
- ❌ Limited in terms of polish and performance

🎵 What a Rhythm Game Needs

- 🎧 **Beat Detection or Timing System** (e.g., hit notes on the beat)
- 🎮 **Input Sync** (e.g., player presses buttons in time)
- 📊 **Scoring System**
- 🎬 **Visual Cues** (e.g., arrows, circles, or notes moving)
- 🕒 **Timing Accuracy** (Perfect, Good, Miss)

Game engines make all this much easier by providing audio timing APIs and visual tools.

🧑‍💻 Alternative (but harder) Routes

If you *don't* use a game engine, here's what you'll face:

1. Custom Code with [Pygame](https://www.pygame.org) (Python)

- 🎯 Doable, but you'll write your own audio timing, input, collision, etc.
- Good for learning, but **not beginner-easy**

2. Web-based Rhythm Game

- Use JavaScript + HTML5 Canvas
- Libraries like Tone.js or Howler.js for audio timing
- Not as beginner-friendly unless you're already into web dev

---

✅ Recommendation

Start with **Godot** or **Unity**:

- Godot is simpler for pure 2D, less setup.
- Unity is more powerful, especially long-term.

Once you're more experienced, you can explore custom implementations.

To run a **9-directional dance pad** that connects to a **PC via USB** and also communicates with an **LED driver**, you'll need a microcontroller (MCU) that supports:

✅ Core Requirements:

1. **Enough GPIOs**:
   - At least **9 digital inputs** for the pad sensors.

    - Additional **digital outputs** (or a serial interface) to control the LED driver.
2. **USB Device Capability**:
    - Native USB HID or CDC support (for PC connection as a game controller or serial device).
3. **LED Driver Communication**:
    - Either GPIOs (for basic on/off) or a protocol like **SPI**, **I2C**, or **UART** to communicate with an LED controller.
4. **Low Latency**:
    - For responsive input and visual feedback.

💡 Recommended MCU Options:

🔷 Teensy 4.0 or 4.1

- **Pros**:
    - Native USB support (appears as joystick, keyboard, or HID).
    - Powerful ARM Cortex-M7 @ 600 MHz (overkill but very low latency).
    - Tons of GPIOs and hardware serial, SPI, and I2C.
    - Great community support.
- **Cons**:
    - Slightly more expensive than basic MCUs.
- **Best for**: High-performance setups with RGB matrix or advanced lighting.

🔷 STM32F103C8T6 (Blue Pill)

- **Pros**:
    - Native USB support (with STM32duino or libusb).

    - Multiple GPIOs, SPI/I2C/UART.
    - Very affordable.
- **Cons**:
    - Trickier USB HID setup than Teensy.
- **Best for**: DIY USB devices with moderate learning curve.

🔷 **Arduino Leonardo / Micro**

- **Pros**:
    - Native USB HID support (built-in joystick/keyboard emulation).
    - Simple to program.
    - Enough GPIOs.
- **Cons**:
    - 8-bit AVR (slower than ARM).
    - Limited if you want more complex LED animations.
- **Best for**: Simple, affordable dance pad setups.

🔷 **Raspberry Pi Pico**

- **Pros**:
    - RP2040 chip with dual-core.
    - USB HID support via CircuitPython or C++ SDK.
    - SPI/I2C/UART support.
- **Cons**:
    - Somewhat more setup time to configure USB HID.
- **Best for**: Balanced performance and cost, open source.

⚡ LED Driver Options

If using individually addressable LEDs (e.g. WS2812, APA102):

- Use **NeoPixel** or **FastLED** libraries.
- These require **timing-accurate GPIO** (Teensy is great here).

If using a basic LED panel (on/off for each pad):

- Use GPIOs or an **I2C/SPI GPIO expander** (e.g. **MCP23017**, **74HC595**).

🎮 USB HID vs Serial

- If you want it to act like a **game controller**, use **USB HID Joystick** profile (Teensy/Leonardo are ideal).
- If you want it to send messages to a **custom PC app**, use **USB Serial (CDC)** and parse inputs.

🔧 Final Recommendation

If you want **plug-and-play USB controller functionality**, responsive LED control, and easy development:

> ✅ **Teensy 4.0** or **Arduino Leonardo**

If you want more power and flexibility (with some DIY setup):

> ✅ **Raspberry Pi Pico** or **STM32F103 (Blue Pill)**

Let me know:

- Do you want analog pressure sensitivity?

- What kind of LED panel or effects do you want?
- Should it plug in and work like a controller right away?

ix

That will help narrow it down further.

For arcade-grade and DIY dance pads, **the most commonly used and effective panel sensor types** are:

🔹 1. Mechanical Switch-Based Sensors (Arcade Standard)

🟢 **Best for: Arcade-grade dance pads (like DDR, ITG)**

- **Used in:** Original DDR arcade pads (Konami), ITG pads, StepManiaX
- **How it works:** Each panel is mounted over a set of mechanical switches (typically 4), and stepping on the panel compresses it to trigger the switches.
- **Pros:**
  - Extremely durable under heavy use
  - High responsiveness and low latency
  - Precise actuation point (clear "on" and "off")
- **Cons:**
  - Requires fine calibration and tuning
  - Panels can wear out or misalign over time if not maintained
- **Common switch:** Omron V-10 or V-15 series

🔹 2. Foil Contact Sensors

🟡 **Best for: DIY dance pads on a budget**

- **Used in:** Cobalt Flux-style DIY pads, older homemade builds

- **How it works:** Aluminum foil or copper tape layers make contact when stepped on, completing a circuit.
- **Pros:**
  - Very inexpensive and easy to build
  - No need for expensive mechanical parts
- **Cons:**
  - Less consistent triggering
  - Prone to wear, corrosion, or false positives
  - Not suitable for heavy or high-performance use

🔹 3. Force-Sensitive Resistors (FSRs)

🟡 **Best for: Intermediate DIY pads**

- **Used in:** Some high-end DIY projects
- **How it works:** Measures pressure applied to the sensor—more pressure = lower resistance.
- **Pros:**
  - Analog input – allows pressure sensitivity
  - Flexible placement options
- **Cons:**
  - More expensive than foil
  - May need analog-to-digital conversion (ADC) circuitry
  - Less tactile feel than switches

🔹 4. Capacitive Touch or IR Proximity (Rare)

🔴 **Not recommended for dance pads**

- **Used in:** Experimental or futuristic pads
- **Cons:**
  - Not designed for physical stomping
  - Too sensitive or unreliable for high-intensity gameplay

✅ Recommendation Summary

Use Case	Recommended Sensor	Why
Arcade-quality	Mechanical switches	High durability, precision, matches arcade behavior
DIY (budget)	Foil contact sensors	Cheap, easy to build, good for light/moderate use
DIY (premium)	FSRs or mechanical	Smooth, analog response; mechanical for better feel

---

Let me know your build goals (e.g. USB connectivity, LED response, number of panels, budget), and I can suggest a full sensor + controller setup for your pad.

[24] Microsoft. (2025). copilot [AI assistant]. Microsoft Corporation.

What would be the best way to create a rhythm game for a beginner? Would using a game engine be best or are there other alternatives to complete this objective?

🎮 Great question, Andres! Creating a rhythm game can be a super fun way to dive into game development, especially if you're into music and timing. Whether you want to build something simple like *Tap Tap Revenge* or more intricate like *Friday Night Funkin'*, here's how you could get started:

🛠️ Best Options for a Beginner

1. **Using a Game Engine** (Highly Recommended!)

Game engines handle graphics, sound, input, and more — all the essentials for a rhythm game.

- **Unity** (with C#)
    - One of the most popular engines for rhythm games.
    - Tons of community tutorials and asset store packages.
    - Great audio timing tools via *AudioSource* and *Timeline*.
- **Godot** (with GDScript)
- Lightweight and beginner-friendly.
- Has a built-in animation player and audio support.
- Open-source with lots of documentation.

2. **No-Code or Low-Code Tools**

If coding feels overwhelming, these tools let you focus more on design and logic.

- **Construct 3** or **GDevelop**
- Drag-and-drop interfaces.
- You can create rhythm-based actions with timers and animations.
- Perfect for prototyping without writing code.

3. **Web-Based Tools**

Want to build something playable in a browser?

- **p5.js** (JavaScript-based creative coding library)
    - Lets you sync visuals to audio beats.
    - Requires some coding, but great for creative freedom.
- **Scratch**
- Ultra beginner-friendly.
- You can make simple rhythm games with sprite animations and sound blocks.

🎵 Core Elements of a Rhythm Game

Here's what you'll need to think about, regardless of the tool:

- Beat detection or fixed timing system
- Input controls (keyboard, touchscreen, etc.)
- Visual cues (like notes or arrows)
- Sound synchronization
- Scoring system

🚀 Kickstart Suggestions

- Try recreating a tiny version of an existing rhythm game — like one level with simple visuals and sounds.
- Use free music tracks or sound effects to experiment.

- Focus on getting the timing and feel right — rhythm games live and die by how satisfying the feedback is.

If you tell me what kind of theme or vibe you're thinking for the game, I can help brainstorm mechanics or even give you a starter outline to build from. Got a favorite rhythm game you're trying to emulate?

For a 9-directional dance pad that connects to a PC via USB and also controls an LED panel, you'd want a microcontroller unit (MCU) that supports:

- ✅ *USB HID (Human Interface Device) for PC input recognition*
- ✅ *Enough GPIO pins for 9 directional inputs*
- ✅ *Communication protocols like SPI or I2C to control an LED driver*
- ✅ *Good community support and libraries for ease of development*

Here are some solid MCU options:

🧠 ***Recommended MCUs***

MCU Model	***USB Support***	***GPIO Pins***	***LED Driver Communication***	***Notes***
Arduino Leonardo	*Native USB HID*	*~20*	*SPI/I2C*	*Easy to use, great for beginners*
Teensy 4.0	*Native USB HID*	*~40*	*SPI/I2C*	*Very fast, compact, and powerful*
Raspberry Pi Pico	*USB HID via TinyUSB*	*~26*	*SPI/I2C*	*Affordable and flexible with RP2040 chip*
STM32F103 ("Blue Pill")	*USB HID via libraries*	*~37*	*SPI/I2C*	*More advanced, great performance*

🔌 ***LED Driver Compatibility***

*If you're using something like the **BCT3119 9-channel constant-current LED driver**, which uses SPI for control, all the above MCUs can communicate with it via SPI. You'll just need to ensure voltage compatibility (most are 3.3V or 5V).*

🕹️ ***Dance Pad Input***

Each directional pad switch can be wired to a digital input pin. Use internal pull-up resistors and debounce logic in software to ensure clean input signals.

🖥️ ***PC Integration***

For USB connection to the PC:

- ***Arduino Leonardo** and **Teensy** natively support USB HID, so they can emulate a keyboard or game controller.*
- ***Raspberry Pi Pico** can use the [TinyUSB library](#) to emulate HID devices.*
- ***STM32** requires more setup but is highly customizable.*

Would you like help wiring the pad or writing the firmware logic for input and LED control?

*For arcade-grade and DIY dance pads, the **type of panel sensor** you choose is crucial for responsiveness, durability, and overall gameplay experience. Here's a breakdown of the best options:*

🥇 *Top Sensor Types for Dance Pads*

1. Conductive Plate Sensors (Arcade Style)

- ***Used in original DDR and Pump It Up arcade machines***
- *Two metal plates separated by foam or spring material*
- *Pressure from a footstep compresses the gap, completing the circuit*
- ✅ *Highly responsive and durable*
- 🔧 *Requires precise construction and materials*

2. FSR (Force Sensitive Resistor) Sensors

- *Detect pressure changes rather than direct contact*
- *Often paired with microcontrollers like Arduino*
- ✅ *Great for custom sensitivity tuning*
- ⚠️ *May require calibration and software setup*
- *[FSR pad guide](#) offers excellent DIY instructions*

3. Chinese Arcade Sensors (AliExpress)

- *Affordable and widely used in DIY builds*
- *Typically black rubber sensors with JST connectors*
- ✅ *Easy to install and replace*
- 💬 *Highly recommended by DIY builders on Reddit and Instructables*

4. 3D Printed Spring Sensors

- *Innovative DIY option using flexible plastic as the spring*
- *Copper tape used for conductivity*
- ✅ *Customizable and cost-effective*
- ⚠️ *Long-term durability depends on material choice (PLA vs. Nylon)*
- *[Detailed build guide](#) available*

🛠️ *Recommended for DIY Builders*

- *If you're aiming for arcade-like feel:* ***Conductive plate sensors*** *or* ***Chinese arcade sensors***
- *If you want customization and tech integration:* ***FSR sensors***
- *For a budget-friendly and creative build:* ***3D printed spring sensors***

Would you like help choosing materials or designing your own pad layout? I can walk you through a build plan or help source parts.

[25] Gemini. (2025, July 6). [Response to a query about rhythm game development]

What would be the best way to create a rhythm game for a beginner? Would using a game engine be best or are there other alternatives to complete this objective?

For a beginner looking to create a rhythm game, using a game engine is generally the ***best and most recommended approach****. While alternatives exist, game engines offer significant advantages in terms of ease of use, built-in features, and community support, which are crucial for someone starting out.*

Here's why game engines are ideal and what alternatives exist:

Why Game Engines are Best for Beginners

Game engines provide a framework and tools that simplify many complex aspects of game development, especially for rhythm games:

- ***Audio Synchronization:*** *Rhythm games are all about precise timing with music. Game engines have robust audio systems that make it much easier to load, play, and synchronize events with audio tracks. Many even offer tools for beat mapping or have plugins that help analyze audio to extract beat information.*
- ***Visuals and Animations:*** *Creating the visual feedback for hits, note scrolling, and character animations is streamlined with an engine's visual editors and animation tools.*
- ***Input Handling:*** *Game engines simplify capturing player input (key presses, taps, etc.) and checking them against timed events.*
- ***Physics (if needed):*** *While less critical for core rhythm game mechanics, if you want any physical interactions (e.g., objects reacting to hits), the engine's physics engine will handle it.*
- ***UI Development:*** *Most engines have intuitive UI systems for creating scores, combo meters, health bars, and menus.*
- ***Cross-Platform Deployment:*** *If you eventually want your game on different platforms (PC, mobile, web), engines often have built-in tools for exporting to multiple targets.*

* ***Community and Resources:*** *Popular game engines have vast communities, tutorials, forums, and asset stores, which are invaluable for learning and troubleshooting.*

Recommended Game Engines for Rhythm Games (Beginner-Friendly)

1. ***Godot Engine:***
    * ***Pros:*** *Free and open-source, lightweight, excellent for 2D games (which is common for rhythm games), has its own beginner-friendly scripting language (GDScript, similar to Python), and a growing community.*
    * ***Considerations:*** *While its audio features are good, you might need to dive a bit into scripting for advanced synchronization.*
2. ***Unity:***
    * ***Pros:*** *Extremely versatile, widely used, massive community and asset store, robust audio tools, and a good choice for both 2D and 3D.*
    * ***Considerations:*** *Can be a bit more overwhelming at first due to its vastness, uses C# for scripting (which is a good language to learn, but might be a slightly steeper curve than GDScript for absolute beginners).*
3. ***GameMaker Studio 2:***
    * ***Pros:*** *Very beginner-friendly, especially if you prefer a visual scripting approach (though it also has its own scripting language, GML), excellent for 2D games, and a good option for quickly prototyping ideas.*
    * ***Considerations:*** *Less powerful for complex 3D games compared to Unity or Unreal.*

Key Considerations for Rhythm Game Development

Regardless of the engine, here are some core concepts you'll need to grasp:

* ***Beat Mapping/Charting:*** *This is the process of defining when notes or actions should occur in relation to the music. You'll likely need a system (even a simple one) to "chart" your songs.*
* ***Synchronization:*** *This is the most critical aspect. You need to ensure visual elements and input windows are perfectly aligned with the audio. Factors like audio latency need to be accounted for.*
* ***Input Window/Timing:*** *Define the "perfect hit" window and "good hit" windows around each beat.*
* ***Visual Feedback:*** *Players need clear visual cues for upcoming notes and immediate feedback for their actions (e.g., a glow on hit, a miss indicator).*
* ***Scoring and Combos:*** *Implement a system to track player performance, scores, and combo streaks.*

Alternatives to Game Engines

While not recommended for a beginner building their first game, these alternatives exist:

- ***Building from Scratch with Programming Languages and Libraries:***
    - ***Description:*** *This involves writing all the game logic, rendering, audio playback, and input handling yourself using a programming language (like Python with Pygame, JavaScript with a canvas library, or C++ with SDL/SFML).*
    - ***Pros:*** *Complete control over every aspect, deep learning experience in programming.*
    - ***Cons:*** *Significantly more complex and time-consuming for a beginner. You'd have to implement basic engine features yourself, which can be a huge hurdle. This is generally only recommended if your primary goal is to learn low-level game programming rather than making a game quickly.*
- ***No-Code/Low-Code Platforms (less common for rhythm games):***
    - ***Description:*** *Tools like Construct 3 or GDevelop allow you to create games with visual scripting and drag-and-drop interfaces, often without writing traditional code.*
    - ***Pros:*** *Extremely beginner-friendly, can create simple games very quickly.*
    - ***Cons:*** *May lack the fine-grained control and advanced audio synchronization capabilities needed for complex rhythm games. While you might be able to create a very basic "Simon Says" style rhythm game, something like Guitar Hero would be challenging.*

In summary, for a beginner, a game engine like Godot or Unity is by far the best way to start creating a rhythm game. *They provide the necessary tools and support to focus on the unique challenges of rhythm game development without getting bogged down in low-level programming complexities. Start with a simple concept, maybe just a single note type and a short song, and build up from there.*

For a 9-directional dance pad with USB connectivity to a PC and LED panel control, you'll need an MCU that offers:

1. ***USB HID (Human Interface Device) Emulation:*** *This is crucial for the dance pad to be recognized by the PC as a gamepad or joystick without requiring custom drivers.*
2. ***Sufficient GPIO Pins:***
    - ***For the 9-directional pads:*** *Each pad will act as a digital input. You'll need at least 9 input pins, and potentially more if you use a matrix scanning approach to reduce pin count.*
    - ***For LED Panel Control:*** *The number of pins required depends on the type of LED panel.*

        - ***Addressable LEDs (e.g., WS2812B, NeoPixels):*** *These are ideal as they only require one data pin for a whole string of LEDs, greatly simplifying wiring. You'd typically need one data pin per LED strip (e.g., if you have a strip per pad, you'd need 9 data pins).*
        - ***Non-addressable LEDs (individual LEDs or common anode/cathode strips):*** *These will require more GPIOs, possibly using shift registers (like the 74HC595) or dedicated LED driver ICs (like the TLC5940) to expand the output pins from the MCU.*
3. ***Processing Power (moderate):*** *While a dance pad doesn't require immense processing power, the real-time input sensing and LED control will benefit from a decent clock speed.*
4. ***Memory:*** *Enough memory for your firmware, including debouncing logic for the pad inputs and potentially LED animation patterns.*

Here are some strong MCU candidates, keeping in mind ease of use and community support:

1. Arduino Leonardo / Arduino Micro / Adafruit ItsyBitsy M0/M4 (or similar SAMD21/SAMD51 boards):

- ***Pros:***
    - ***Native USB HID:*** *These boards are excellent for emulating USB devices like keyboards or joysticks directly, which is perfect for a dance pad. The Arduino Leonardo is specifically cited in DIY dance pad projects for its HID capabilities.*
    - ***Easy to program:*** *Arduino IDE and its extensive libraries make programming very accessible.*
    - ***Sufficient GPIOs:*** *The Leonardo has 20 digital I/O pins, and boards like the ItsyBitsy M4 have even more (e.g., 25 digital GPIOs), which should be enough for 9 inputs and several LED control lines (especially with addressable LEDs or external drivers).*
    - ***Good for LED control:*** *Easily integrates with FastLED library for addressable LEDs. For non-addressable LEDs, you can use I2C/SPI-controlled LED drivers.*
- ***Cons:*** *Might be slightly more expensive than some bare microcontrollers, but the development ease often outweighs this.*

2. ESP32 / ESP32-S3:

- ***Pros:***
    - ***Built-in Wi-Fi and Bluetooth:*** *While not strictly necessary for your stated requirements, this opens up possibilities for wireless connectivity in the future, if desired (e.g., for data logging or remote control).*
    - ***Plenty of GPIOs:*** *ESP32 boards typically have a good number of GPIOs (around 30-40 depending on the specific module).*
    - ***Fast processing:*** *Dual-core processors in some ESP32 models provide ample power.*

    - ***USB-to-Serial converter built-in:*** *Many ESP32 boards have a CP2102 or similar chip, making uploading code easy. Some newer ESP32-S3 boards even offer native USB for direct HID emulation, like the Arduino options.*
    - ***Good for LED control:*** *Excellent support for FastLED and other LED libraries.*
- ***Cons:*** *Might have a slightly steeper learning curve than basic Arduino boards if you're completely new to microcontrollers, but the community is huge.*

3. Raspberry Pi Pico / Pico W:

- ***Pros:***
    - ***Low Cost:*** *Very affordable.*
    - ***RP2040 Microcontroller:*** *Designed by Raspberry Pi, it's a powerful chip for its price point.*
    - ***Programmable I/O (PIO):*** *This is a unique feature that allows for highly precise and flexible control over I/O, which could be useful for advanced LED patterns or precise input timing.*
    - ***USB MicroPython/CircuitPython/C++:*** *Supports multiple programming environments, including CircuitPython which makes USB HID incredibly straightforward.*
    - ***Good GPIO count:*** *Sufficient pins for your needs.*
- ***Cons:*** *While powerful, the ecosystem might feel a bit less mature for beginners compared to Arduino, though CircuitPython bridges this gap very well.*

Key Considerations for your project:

- ***Input Method for the Dance Pad:***
    - ***Individual Switches:*** *Each of the 9 directions (and potentially a center button) can be a simple momentary switch. You'll need debouncing in your code to prevent multiple readings from a single press.*
    - ***Matrix Scanning:*** *If you want to reduce the number of GPIOs, you can arrange your switches in a matrix (rows and columns). The MCU would then scan the matrix to detect pressed buttons. This is more complex to implement in code but saves pins. Given 9 directions, individual switches are likely manageable.*
- ***LED Panel Control:***
    - ***Addressable LEDs (WS2812B/NeoPixels):*** *Highly recommended. You connect the data line of your LED strip to a single GPIO pin, and then you can control each LED's color and brightness individually with simple code. This is much simpler than wiring up individual LEDs or complex matrices.*
    - ***Non-addressable LEDs with drivers/shift registers:*** *If your LED panel uses standard non-addressable LEDs, you'll need external components like MOSFETs for power switching and potentially shift registers or dedicated LED driver ICs (e.g., TLC5940, MAX7219) to control many LEDs with fewer MCU pins.*

- ***Power Supply:*** *Consider the power requirements of your LED panel. If it's a large panel, it will likely need its own dedicated power supply, and you'll switch the power on/off through the MCU using a relay or MOSFET.*

Recommendation:

For a good balance of ease of use, robust USB HID, and sufficient GPIOs for both input and LED control, an ***Arduino Leonardo*** *or an* ***Adafruit ItsyBitsy M0/M4 (or similar SAMD21/SAMD51 based board)*** *would be excellent choices, especially if you plan to use addressable LEDs. They offer straightforward programming and great community support for HID emulation. If you want to experiment with wireless capabilities or a slightly more powerful platform for future expansion, the* ***ESP32-S3*** *(with native USB) or* ***Raspberry Pi Pico W*** *are also strong contenders.*

For both arcade-grade dance pads and DIY dance pads, the best type of panel sensor often comes down to a balance of ***durability, sensitivity, and ease of implementation.***

Here's a breakdown of the most common and recommended sensor types:

1. Force-Sensing Resistors (FSRs)

Why they are often considered the best, especially for DIY and modern arcade pads:

- ***Sensitivity and Precision:*** *FSRs change their resistance based on the pressure applied, allowing for very precise and customizable sensitivity. This means you can finely tune how much force is needed to register a step. Modern pads like the StepManiaX Generations 4+ use FSRs.*
- ***Thin Profile:*** *They are incredibly thin, making them easy to integrate into existing pad structures or incorporate into DIY builds without significantly altering the panel height or feel.*
- ***DIY Friendly:*** *FSRs are relatively straightforward to wire up to microcontrollers like Arduino or Teensy boards. Many online guides and firmware options exist specifically for FSR-based dance pads.*
- ***Individual Sensor Thresholds:*** *With FSRs, each sensor's sensitivity can be adjusted individually in software, which is great for personalizing the pad's response and reducing issues like double-triggering from vibrations.*
- ***Cost-Effective:*** *While not as cheap as some rudimentary DIY methods, FSRs are generally affordable for the performance they offer.*

Considerations:

- ***Durability (can vary):*** *While robust, care is needed during installation to prevent bending or kinking, which can affect their lifespan. Proper mounting and protection are important.*

- ***Non-linear Response:*** *Their resistance change can be non-linear, which requires careful calibration in software to achieve consistent sensitivity across the pressure range.*
- ***Drifting:*** *Some users report that FSRs can be prone to "drifting" over time, meaning their readings might subtly change, requiring occasional recalibration.*

2. Arcade Sensors (Rubber Contact Sensors with L-Brackets)

The traditional choice for official arcade machines:

- ***Durability:*** *These sensors are designed to withstand years of heavy stomping in arcade environments. They typically involve a rubber strip with an air bladder and metal L-brackets that press down to complete a circuit.*
- ***Reliability:*** *When properly maintained, they offer consistent and reliable input.*
- ***Authentic Feel:*** *Many players prefer the distinct "feel" of genuine arcade sensors.*

Considerations:

- ***Cost:*** *Genuine arcade sensors can be quite expensive, especially if sourced new. Reproduction sensors are available and more affordable.*
- ***Complexity of Installation:*** *Retrofitting them into a DIY pad can be more involved as they require specific channels and L-brackets for proper actuation.*
- ***Maintenance:*** *They often require padding (tape, paper) under or above the L-bracket to adjust sensitivity and maintain responsiveness over time.*

3. Contact Sensors (DIY Metal Plates/Foil)

A common and inexpensive DIY method:

- ***Simplicity and Low Cost:*** *This method involves two conductive surfaces (like aluminum foil or metal plates) separated by a compressible material (like foam or weather stripping). When pressed, they make contact and complete a circuit.*
- ***Easy to Understand:*** *The concept is very basic, making it a popular starting point for DIYers.*

Considerations:

- ***Durability:*** *Aluminum foil can tear easily, and other materials may degrade over time, leading to inconsistent performance.*
- ***Sensitivity Issues:*** *Achieving consistent and precise sensitivity can be challenging. "Penny modding" is a common technique to increase sensitivity in pads using this principle (like some LTEK pads).*
- ***Bounce/Double Triggers:*** *The mechanical nature of contact sensors can sometimes lead to "bouncing" or double-triggering, where a single step registers multiple inputs. This often requires debouncing in software.*

Less Common/Emerging Options:

- ***Load Cells:*** *These are highly accurate force sensors, often used in industrial applications. Some high-end DIY pads or custom builds might incorporate them for extremely precise pressure detection. However, they are generally thicker and require more complex electronics (amplifiers) to interface with microcontrollers.*
- ***Velostat:*** *A pressure-sensitive conductive material that can be cut into sheets. Some early FSR pads used Velostat, but standard FSR modules are now more commonly preferred for consistency.*
- ***Capacitive Sensors:*** *While explored for dance pads, they can be sensitive to environmental factors (like moisture) and shoe types, making them less ideal for consistent performance.*

Recommendation:

*For **arcade-grade dance pads**, modern designs are increasingly leaning towards **FSRs** due to their precise sensitivity, customizability, and relatively thin profile, while still offering excellent durability with proper implementation. The traditional arcade rubber contact sensors remain a strong, durable, and authentic choice if available and within budget.*

*For **DIY dance pads**, **Force-Sensing Resistors (FSRs)** are generally the **best choice**. They offer an excellent balance of performance, ease of use with microcontrollers (like Arduino), and the ability to customize sensitivity to a high degree, allowing for a near-arcade quality experience without the higher cost or complexity of true arcade sensors. If you're on a very tight budget or just experimenting, simple contact sensors can be a starting point, but be prepared for potential durability and consistency issues.*

[26] DeepSeek (2024), [large language model developed by DeepSeek AI].

> *Creating a rhythm game as a beginner is an exciting challenge! Using a game engine is overwhelmingly the best choice for a beginner. Here's why and how to approach it:*
>
> *Why a Game Engine is Best:*
>
> *Abstraction of Complexity: Engines handle graphics rendering, physics (if needed), input management, audio playback, and cross-platform deployment. You focus on your game logic.*
>
> *Built-in Tools: Visual editors, animation systems, asset managers, and debuggers significantly speed up development.*
>
> *Asset Pipelines: Easily import graphics, sounds, and music.*

Community & Resources: Massive tutorials, documentation, forums, and asset stores specifically for game engines. Crucial for learning.

Rapid Prototyping: Test core mechanics (like hitting notes to the beat) very quickly.

Precision Timing: Engines provide high-resolution timers and frame-independent update loops, essential for rhythm game accuracy.

Recommended Game Engines for Beginners (Rhythm Focus):

Unity (C#):

Pros: Largest community & tutorials, vast asset store, powerful & flexible, excellent 2D & 3D support, good performance. Many rhythm game tutorials exist.

Cons: Can feel bloated for very small projects, newer input system has a learning curve.

Why for Rhythm: Mature audio APIs (AudioSource, AudioClip), coroutines for timing, strong animation tools for note highways/effects. Great for both 2D and 3D rhythm games.

Godot (GDScript - Python-like, C#):

Pros: Lightweight, fast, completely free & open-source, incredibly intuitive scene/node system, excellent 2D support, growing community. GDScript is easy to learn.

Cons: Smaller community/resources than Unity (but growing fast), 3D is capable but less mature than Unity/Unreal, asset store smaller.

Why for Rhythm: Built-in AudioStreamPlayer nodes, strong signal system for event handling (great for hit/miss events), Tween nodes for smooth animations. Very well-suited for 2D rhythm games.

Unreal Engine (Blueprints Visual Scripting, C++):

Pros: Stunning visuals "out of the box," powerful Blueprints visual scripting (great if you struggle with text code), best-in-class 3D.

Cons: Steeper learning curve overall, larger download/overhead, C++ is complex for true beginners, potentially overkill for a simple 2D rhythm game.

Why for Rhythm: Robust audio system, Blueprints can be used for core timing logic, great for ambitious 3D rhythm games (like VR).

Avoid (for a Beginner) / Use With Caution:

Pure Programming Languages (Python/Pygame, JavaScript/HTML5 Canvas, C++/SDL, Java/LibGDX):

Pros: Maximum control, deep learning, lightweight.

Cons: You have to build everything from scratch: rendering shapes/sprites, playing sounds precisely, handling input, managing game states, timing. High risk of getting bogged down in low-level details before making game progress. Audio timing precision is notoriously tricky without engine help.

When Consider? Only if you have significant programming experience already and specifically want the challenge of building the underlying systems. Not recommended for a first project.

Best Approach for a Beginner Using an Engine:

Choose Your Engine: Pick Unity (broadest resources) or Godot (simplicity, great 2D) based on your preference. Download it.

Start TINY: Your first goal is NOT "Guitar Hero Clone." Aim for: "A single button that I have to press when a moving dot hits a line, synchronized to a simple drum beat."

Learn Engine Fundamentals:

Follow the official "Getting Started" tutorials for your engine.

Understand core concepts: Scenes/Objects, Components/Nodes, Transforms, Basic Scripting (C# for Unity, GDScript for Godot).

Core Rhythm Prototype:

Audio: Import a short, simple loop (e.g., a 4-beat drum pattern). Learn to play it.

Note Spawning: Write code to spawn a visual object (a cube, circle) at a specific time before it needs to be hit. Hardcode the timing at first.

Note Movement: Move the spawned object towards a "hit line" at a constant speed. Ensure its arrival time matches the beat time.

Input Detection: Detect when the player presses the designated key (e.g., Spacebar).

Judgment: When the key is pressed, check the distance of the note object to the hit line.

Very close (e.g., within 50ms window): "Perfect!" (Destroy note, add score).

Close (e.g., 50-100ms): "Good!"

Too early/late: "Miss!" (Destroy note, maybe show penalty).

Feedback: Show text ("Perfect!"), play sounds, add particle effects on hit/miss.

Refine & Expand: Once the core loop works:

Level Design: Create a system (like a text file or simple array) to define note timings instead of hardcoding.

Multiple Lanes/Buttons: Add more hit lines and map them to different keys (e.g., Left/Right Arrow, A/S/D keys).

Better Graphics: Replace cubes with sprites, add a background.

UI: Add a start screen, score display, combo counter.

Music: Use a full song instead of a loop. Crucially, learn about audio offset calibration to sync your note timing perfectly with the track in the engine.

Difficulty: Adjust note speed, timing window size, or note density.

Crucial Rhythm Game Specifics:

Precise Timing: Use the engine's high-resolution time (Time.time in Unity, OS.get_ticks_usec() in Godot) for judgment, not frame-based logic. Account for display latency if needed (less critical early on).

Audio Analysis (Advanced): Initially, manually chart your note timings by listening to the song. Later, explore tools or libraries (like BASS, FMOD integration) for beat detection, but manual charting is standard for precise rhythm games.

Input Lag: Be aware that monitors/TVs, audio devices, and peripherals add lag. Calibration settings in your game are essential later on.

Alternatives ONLY if Engines Feel Too Heavy (Still Not Ideal):

Specialized Frameworks:

LÖVE (Lua): Simpler framework than raw OpenGL, but still requires building most systems. Lua is easy to learn.

Phaser (JavaScript): Web-based framework. Good if you know JS and want a web game. Still requires more setup than an engine.

Rhythm Game Creation Tools (Limited): Tools like "K-Shoot Mania Editor" or "StepMania/Etterna Editor" exist, but they are for creating charts for existing, specific rhythm game engines, not building your own unique game from scratch.

Conclusion:

Start with a game engine (Unity or Godot are top recommendations for beginners). Focus relentlessly on building the tiniest possible playable rhythm prototype (one button, one note type, one beat). Learn the engine fundamentals through that process. Resist the urge to build your dream game immediately. Completing a tiny, functional prototype is infinitely more valuable and motivating than getting lost in the complexity of a "from-scratch" approach or an overly ambitious first design. Good luck!

For a **9-directional dance pad** with **USB connectivity** to a PC and **LED control**, you'll need an **MCU** with the following features:

- **USB HID** (for input as a game controller)

- **GPIOs** for button matrix scanning (9 directions + optional extras)

- **Hardware communication** (I^2C, SPI, or UART) for an LED driver (e.g., WS2812B, PCA9685, or TLC5940)

- **Sufficient processing power** for real-time input & LED updates

Best MCU Options

1. Raspberry Pi Pico (RP2040) – Best Budget & Flexible Choice

 - **Pros**:

  - Cheap (~$4-$5)

  - Dual-core Cortex-M0+ (good for multitasking)

  - Built-in USB (can act as HID device)

  - Many GPIOs (supports button matrix + LED driver)

  - Easy CircuitPython/MicroPython/C++ support

 - **Cons**:

  - Requires manual USB HID setup (but libraries exist)

 - **Best for**: DIY projects where cost matters

2. Arduino Leonardo/Pro Micro (ATmega32U4) – Best Plug-and-Play USB HID

- **Pros**:
  - Native USB HID (recognized as a keyboard/joystick instantly)
  - Works well with button matrices
  - Can control LEDs via I^2C/SPI (e.g., using FastLED for addressable LEDs)
- **Cons**:
  - Limited RAM/Flash (but enough for this project)
- **Best for**: Quick USB game controller setup

3. ESP32-S3 (Native USB) – Best for Wireless + Advanced Features

- **Pros**:
  - Native USB (can act as HID)
  - Wi-Fi/Bluetooth (if you want wireless options later)
  - Fast, dual-core, plenty of GPIOs
- **Cons**:
  - Slightly more expensive (~$10)
- **Best for**: Future-proofing (if you want wireless later)

4. STM32 Blue Pill (STM32F103) – Best for High Performance

- **Pros**:
  - Fast Cortex-M3, lots of peripherals
  - Can do USB HID with some setup
- **Cons**:
  - Requires more firmware work

- **Best for**: Advanced users who need speed

Recommended Setup

1. **MCU**: **Raspberry Pi Pico** (best balance of price & features)

2. **Button Input**: 3x3 matrix (9 switches) + optional debouncing

3. **LED Driver**:

   - For addressable LEDs (WS2812B), use FastLED or NeoPixel libraries

   - For PWM control (e.g., PCA9685), use I^2C

4. **USB**: Configure as HID joystick (CircuitPython/Arduino libraries available)

Alternative Approach

If you want **plug-and-play USB**, an **Arduino Pro Micro (ATmega32U4)** is the easiest since it natively supports HID.

Would you like a basic wiring diagram or code examples for any of these?

For **arcade-grade dance pads** and high-quality **DIY dance pads**, the best sensor type is typically a **FSR (Force-Sensitive Resistor)** or a **conductive foam-based contact sensor**, depending on durability, sensitivity, and cost requirements.

Best Sensor Types for Dance Pads:

1. **FSR (Force-Sensitive Resistor)**

   - **Pros:**

     - Highly sensitive to pressure (works with light to heavy steps).

     - Durable (long lifespan, resistant to wear).

     - Used in premium arcade pads (e.g., DDR official cabinets).

   - **Cons:**

- More expensive than other options.

- Requires proper wiring and conditioning circuits.

*2. **Conductive Foam + Metal Contacts (Arcade-Style)***

*- **Pros:***

- Used in original arcade machines (e.g., DDR, PIU).

- Provides a tactile "click" feel when stepped on.

- Reliable and long-lasting when built correctly.

*- **Cons:***

- Requires precise alignment and maintenance.

- Can wear out over time (foam compresses).

*3. **Velostat-Based Pressure Sensors (DIY-Friendly)***

*- **Pros:***

- Cheap and easy to make at home.

- Works by pressure changing resistance.

*- **Cons:***

- Less consistent than FSRs.

- Wears out faster.

*4. **Mechanical Microswitches (Less Common)***

*- **Pros:***

- Tactile feedback (clicky response).

- Durable if high-quality switches are used.

*- **Cons:***

- Requires significant force to activate.

- Can feel stiff compared to arcade pads.

*### **Best Choice?***

*- **For arcade-grade pads:** **FSR** or **conductive foam + metal contacts** (most authentic).*

*- **For DIY builds on a budget:** **Velostat-based sensors** or **soft potentiometers**.*

*- **For a balance of cost and quality:** **FSRs** (best for serious players).*

Would you like recommendations for specific wiring setups or suppliers for these sensors?

How can we implement computer vision into a dance rhythm game using a camera?

ChatGPT said:

Implementing computer vision in a dance rhythm game using a camera involves tracking and interpreting the player's body movements to determine whether they match specific dance poses or actions required by the game. Here's a structured overview on how to implement this:

🎯 Goal

Use a camera to **track player movements** and compare them to **expected dance moves** in sync with rhythm/music.

🧱 Core Components

1. Camera Input

- Use a webcam or phone camera to capture real-time video of the player.

2. Pose Estimation

- Use a **pose estimation model** to detect body landmarks (key points like joints).
    - Popular libraries:
        - MediaPipe Pose
        - OpenPose
        - MoveNet
- Output: 2D or 3D coordinates of body joints (e.g., wrists, elbows, knees, etc.).

3. Dance Move Definition

- Define expected poses or movements at specific timestamps (like "Step left" at beat 3).
    - Use:
        - Pre-defined skeleton positions (keypoint sets).
        - Motion sequences.
        - Tolerance thresholds for accuracy.

4. Real-Time Matching

- Compare the player's live pose with the target pose:
    - Compute a similarity score (e.g., cosine similarity or Euclidean distance between keypoints).
    - Use Dynamic Time Warping (DTW) or optical flow for matching sequences over time.

5. Scoring System

- Assign points based on:
    - Pose accuracy.

- Timing (sync with beat).
- Smoothness of motion.

6. Rhythm Synchronization

- Sync music/audio cues with target moves using beat detection or predefined timestamps.

7. User Feedback

- Provide real-time visual or audio feedback like:
  - "Perfect", "Good", "Miss"
  - On-screen skeleton overlay for correction.

🧪 Technologies & Tools

Task	Tool/Library
Camera capture	OpenCV, MediaPipe
Pose estimation	MediaPipe, OpenPose, MoveNet
Real-time feedback	Pygame, Unity + Python bridge, Flutter with platform channels
ML backend (if needed)	TensorFlow / PyTorch

Music sync	pydub, librosa, Unity Audio Engine

🧠 Example Architecture (Python)

python

Copy code

```python
import cv2

import mediapipe as mp


mp_pose = mp.solutions.pose

pose = mp_pose.Pose()


cap = cv2.VideoCapture(0)


while cap.isOpened():

    success, frame = cap.read()

    if not success:

        break


    # Convert the image to RGB

    image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    results = pose.process(image)
```

```
    # Draw landmarks
    if results.pose_landmarks:
        mp.solutions.drawing_utils.draw_landmarks(
                                 frame,  results.pose_landmarks,
mp_pose.POSE_CONNECTIONS
        )

    cv2.imshow('Dance Pose Tracker', frame)
    if cv2.waitKey(5) & 0xFF == 27:
        break

cap.release()
```

🕹️ Game Flow Example

1. Load song and predefined move chart.
2. Start camera and detect player pose.
3. At each beat window:
    - Compare player's pose to expected pose.
    - If match within threshold, give score.
4. Update visuals and feedback.

⚠️ Challenges

- **Latency**: Ensure low-lag pose detection.
- **Lighting conditions**: Poor lighting can reduce accuracy.
- **Occlusion**: Some body parts may be hidden.
- **User positioning**: Require calibration/setup step to ensure full body is visible.

🔄 Optional Enhancements

- Use a **mobile device as a camera** (streaming to desktop game).
- Allow **custom choreography creation**.
- Use **AI dance feedback** ("lift arm higher", "face forward").
- Add **multiplayer** mode (track multiple people via pose ID).

Let me know your development environment (e.g., Unity, Python, Flutter), and I can tailor the code or tools accordingly!

How can we implement computer vision into a dance rhythm game using a camera?

Implementing computer vision into a dance rhythm game using a camera involves several key steps to track a player's movements and compare them against target dance moves. Here's a breakdown of how it can be achieved:

1. Real-time Pose Estimation:

- **The Core Technology:** The most crucial aspect is real-time human pose estimation. This involves identifying and tracking key points (landmarks) on a person's body (e.g., head, shoulders, elbows, wrists, hips, knees, ankles) from a live camera feed.
- **Libraries and Frameworks:** Several open-source computer vision libraries are excellent for this:
    - **MediaPipe Pose:** Developed by Google, MediaPipe offers a robust and efficient solution for real-time pose estimation. It can track up to 33 2D or 3D landmarks on the human body. It's well-suited for interactive applications due to its performance.

    - **OpenPose:** Another popular and powerful library for multi-person 2D pose estimation. While it might be more resource-intensive, it offers high accuracy.
    - **YOLO (You Only Look Once) with Pose Estimation:** YOLO is primarily an object detection model, but variations exist that incorporate pose estimation capabilities. You might need to train or fine-tune a model for optimal dance-specific pose detection.
- **How it works:** These libraries use deep learning models to analyze each frame of the video feed, locate the human body, and then identify the precise coordinates of the key points.

2. Data Acquisition and Processing:

- **Camera Input:** The game will need to capture video frames from a standard webcam or a more advanced depth camera (like a Kinect, although newer, software-based solutions are often sufficient).
- **Frame-by-Frame Processing:** Each frame from the camera needs to be fed into the pose estimation model.
- **Keypoint Extraction:** The output of the pose estimation will be a set of coordinates for each detected body landmark. These coordinates will be the raw data representing the player's pose at that moment.

3. Reference Dance Data:

- **Pre-recorded Choreography:** You'll need reference data for each dance move or sequence. This can be obtained by:
    - **Motion Capture:** Using professional motion capture suits to record a dancer's movements with high precision. This is ideal for accuracy but can be expensive.
    - **Video Analysis:** Recording a skilled dancer performing the moves and then using the same pose estimation techniques to extract their keypoint data. This is a more accessible and common approach for rhythm games.
    - **Manual Annotation:** While time-consuming, you could manually annotate key points in reference video frames, though this is generally less practical for entire dance routines.
- **Data Representation:** The reference dance data should be stored in a similar format to the player's real-time keypoint data (e.g., a sequence of keypoint coordinates over time).

4. Comparing Player Movement to Reference:

- **Pose Matching/Similarity:** This is the core logic for scoring. You need algorithms to compare the player's current pose (from the live camera feed) to the corresponding reference pose at that moment in the song.
- **Techniques for Comparison:**

- **Euclidean Distance:** Calculate the distance between corresponding key points of the player and the reference. Smaller distances indicate a better match.
    - **Cosine Similarity:** Measure the cosine of the angle between vectors formed by body segments (e.g., upper arm, forearm, thigh, shin). This helps with pose orientation.
    - **Dynamic Time Warping (DTW):** For comparing entire sequences of movements, DTW can be invaluable. It allows for variations in timing and speed, finding the optimal alignment between the player's movement sequence and the reference sequence. This is particularly useful for evaluating fluidity and rhythm over a longer period.
    - **Machine Learning/Deep Learning:** For more complex evaluations and style recognition, you could train a machine learning model to classify poses or evaluate the "correctness" of a move based on a dataset of good and bad performances.
- **Scoring Mechanism:** Based on the similarity metrics, you'll assign a score to the player for each pose or sequence of poses. This score can contribute to the overall game score.

5. Game Logic and Feedback:

- **Rhythm Synchronization:** The game needs to know when each specific dance move is expected to occur in sync with the music. This requires precise timing.
- **Visual Feedback:**
    - **Overlaying Skeletons:** Displaying an overlay of the detected skeleton on the player's live feed can provide immediate visual feedback.
    - **Color-coding:** Highlighting body parts that are correctly or incorrectly positioned.
    - **On-screen Prompts:** Showing visual cues for upcoming moves.
- **Auditory Feedback:** Sound effects for successful moves or missed cues.
- **Scoring Display:** Real-time display of the player's score.

6. Development Considerations:

- **Performance Optimization:** Real-time computer vision can be computationally intensive. You'll need to optimize your code to ensure smooth gameplay, especially for lower-spec hardware. This might involve:
    - **Frame Rate Management:** Processing frames at a consistent rate.
    - **Hardware Acceleration:** Utilizing GPUs for faster processing (e.g., with CUDA or OpenCL).
    - **Model Size and Efficiency:** Choosing pose estimation models that are optimized for speed.
- **Lighting Conditions:** The accuracy of pose estimation can be affected by lighting. Ensure the game's environment is well-lit for optimal performance.
- **Occlusion Handling:** What happens if a player's hand is hidden behind their body? Advanced pose estimation models can sometimes infer occluded joints, but it's a challenge to consider.

- **Calibration:** You might need a calibration step where the player stands in a neutral pose to help the system understand their proportions and position relative to the camera.
- **User Experience (UX):** Make the interface intuitive and the feedback clear.

Example Implementation Flow (Python with MediaPipe and OpenCV):

1. **Install Libraries:** `pip install opencv-python mediapipe`
2. **Initialize Camera:** `cap = cv2.VideoCapture(0)`
3. **Load MediaPipe Pose Model:** `mp_pose = mp.solutions.pose`
4. **Game Loop:**
    - Read frame from camera: `ret, frame = cap.read()`
    - Preprocess frame (e.g., convert BGR to RGB): `frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)`
    - Perform pose estimation: `results = pose.process(frame_rgb)`
    - Extract keypoints: `if results.pose_landmarks:` access `results.pose_landmarks.landmark`
    - Compare to reference data (e.g., load a pre-recorded sequence of landmarks for the current dance step).
    - Calculate score based on pose similarity.
    - Render feedback (e.g., draw landmarks on the frame, display score).
    - Display frame: `cv2.imshow('Dance Game', frame)`
    - Check for exit key.
5. **Release Resources:** `cap.release()`, `cv2.destroyAllWindows()`

By combining powerful pose estimation techniques with careful game design, you can create an engaging and immersive dance rhythm game controlled entirely by a camera.

How can we implement computer vision into a dance rhythm game using a camera?

🕺💻 Using computer vision in a dance rhythm game opens up some exciting gameplay possibilities, especially for hands-free interaction and body-based rhythm tracking. Here's how you can thoughtfully implement it using a camera system:

🧠 Core Concept

Use a camera to track players' body movements in real time, compare them against target poses or dance patterns, and score their performance based on accuracy and timing.

🔧 Key Components to Implement

1. **Pose Estimation**

- **Use models like OpenPose, MediaPipe, or PoseNet to detect and track body joints.**
- **These models process video frames and give you a real-time skeleton map (coordinates of key points like wrists, knees, etc).**

2. **Rhythm & Timing Detection**

- **Synchronize detected body movement events (e.g. arm raises, foot taps) with beat timestamps from the music track.**
- **Introduce a timing window—similar to traditional rhythm games—where player actions are only rewarded if they occur within the beat's "perfect" zone.**

3. **Action Mapping & Scoring**

- **Define expected poses or movement sequences per beat.**
- **Compare detected poses to predefined templates using similarity scores (e.g. cosine similarity between joint vectors).**
- **Implement a scoring system based on timing, accuracy, and difficulty.**

4. **Feedback & Visuals**

- **Provide real-time visual overlays showing detected joints and scoring feedback.**
- **Use skeletal animations or shadow avatars to guide players.**

5. **Calibration & Accessibility**

- **Include a calibration step to accommodate different heights, body types, and camera setups.**
- **Offer multiple difficulty levels and visual guides for different accessibility needs.**

🧪 Prototyping Tools You Might Like

Tool	Use Case	Why It Works Well
MediaPipe	Fast pose detection	Lightweight & cross-platform

Unity + Barracuda	Game dev + ML model integration	Familiar workflow, compatible with C#
OpenCV	Image processing & camera control	Open-source, powerful CV toolkit
Godot + GDNative	Lightweight integration	Great for prototyping with Godot

🎮 **Gameplay Ideas with Computer Vision**

- ***Mirror Match Mode***: Players mimic poses of an on-screen dancer to the beat.
- ***Gesture-to-Rhythm***: Players trigger effects or score by hitting key poses rhythmically.
- ***Freestyle Mode***: AI evaluates style and flow, scores based on creativity + alignment to beat.

If you're thinking of using this in a beginner-friendly engine like Godot, I can help sketch out how to plug in MediaPipe or OpenCV through GDNative or custom modules. Want to go deeper on tech setup or gameplay design next?

Rhythm synchronization in a Godot project can be implemented in several ways, depending on the complexity and purpose (e.g. syncing gameplay events to music beats, rhythm-based input timing, etc.). Below is a general approach to implement rhythm synchronization in Godot (GDScript):

🎵 Step-by-Step: Basic Beat Synchronization System in Godot

---

1. Choose or Load a Music Track

Load the music using an `AudioStreamPlayer` node.

gdscript

CopyEdit

```
# MusicPlayer.gd
extends AudioStreamPlayer


func _ready():
    play()  # Start music
```

---

2. Determine BPM (Beats Per Minute)

You need to know the BPM of your music track. For example, if it's 120 BPM:

gdscript

CopyEdit

```
const BPM = 120
const BEAT_INTERVAL = 60.0 / BPM  # Seconds per beat
```

---

3. Create a Beat Timer

Use a timer to track beats in sync with the song.

gdscript

CopyEdit

```
extends Node


const BPM = 120
const BEAT_INTERVAL = 60.0 / BPM
```

```
var beat_timer := 0.0
var current_beat := 0

onready var music_player := $AudioStreamPlayer

func _ready():
    music_player.play()

func _process(delta):
    beat_timer += delta
    if beat_timer >= BEAT_INTERVAL:
        beat_timer -= BEAT_INTERVAL
        current_beat += 1
        _on_beat()
```

---

4. Define the `_on_beat()` Callback

Trigger anything that should happen on the beat.

gdscript

CopyEdit

```
func _on_beat():
    print("Beat:", current_beat)
    # Call animations, spawn enemies, change colors, etc.
```

---

5. (Optional) Improve Accuracy with `AudioServer` Time

Godot's `AudioServer.get_time_since_playback_start()` provides more precise timing:

gdscript

CopyEdit

```
var last_beat_time := 0.0


func _process(_delta):

    var song_time = AudioServer.get_time_since_playback_start()

    var beat_number = int(song_time / BEAT_INTERVAL)

    if beat_number != current_beat:

        current_beat = beat_number

        _on_beat()
```

---

6. Handling Player Input Timing (For Rhythm Games)

To check if a player's input is on-beat, you can calculate the distance to the nearest beat:

gdscript

CopyEdit

```
func is_input_on_beat(input_time: float, tolerance: float = 0.15) -> bool:

    var beat_number = round(input_time / BEAT_INTERVAL)

    var beat_time = beat_number * BEAT_INTERVAL
```

```
    return abs(input_time - beat_time) <= tolerance
```

Then when input is detected:

gdscript

CopyEdit

```
func _input(event):
    if event.is_action_pressed("hit_note"):
        var current_time = AudioServer.get_time_since_playback_start()
        if is_input_on_beat(current_time):
            print("Good timing!")
        else:
            print("Miss!")
```

🧰 Bonus Tips

- **Use a `.beatmap` or JSON file if you want more complex rhythm patterns synced with music.**
- **Use Godot's `AudioEffectSpectrumAnalyzer` for real-time beat detection (advanced).**
- **If working in Godot 4, `AudioStreamPlayback` and `PlaybackPosition` APIs offer improved control.**
- **Always account for latency—especially on mobile or HTML5 builds.**

Would you like a downloadable rhythm-based mini-game example or help integrating this into your existing project?

Implementing rhythm synchronization in a Godot project, especially for rhythm games, requires precise timing and careful consideration of audio latency. Here's a breakdown of the key concepts and methods you can use:

Core Concepts for Rhythm Synchronization

1. **Audio Playback Position: The most crucial element is accurately knowing the current playback position of your music. Godot's `AudioStreamPlayer` nodes are central to this.**
2. **Latency Compensation: All audio playback has a small, unavoidable delay (latency) between when a `play()` function is called and when the sound is actually heard. This needs to be factored in for precise synchronization.**
3. **BPM (Beats Per Minute) and Time Signature: Understanding the music's tempo and structure is essential for calculating when events should occur.**
4. **Event Scheduling: Instead of relying on `_process()` or `_physics_process()` directly for rhythm events (which can be inconsistent due to varying frame rates), you should schedule events based on the audio's timeline.**
5. **"Conductor" System: A common and effective approach is to create a "conductor" script or node that manages the music's state, calculates beat timings, and emits signals for other game elements to react to.**

Implementation Strategies in Godot

1. Using `AudioStreamPlayer.get_playback_position()` and Latency Compensation

This is the fundamental approach recommended by Godot's official documentation.

- **Get Current Time: You can get the current playback position of an `AudioStreamPlayer` using `audio_stream_player.get_playback_position()`.**
- **Improve Precision with `AudioServer`: For even more precision, especially important in rhythm games, combine `get_playback_position()` with `AudioServer.get_time_since_last_mix()` to get a more real-time position.**

Compensate for Output Latency: Crucially, subtract the `AudioServer.get_output_latency()` from your calculated time. This accounts for the delay between the audio being mixed by the engine and it actually being heard by the player.
GDScript
func _process(delta):

var current_music_time = $AudioStreamPlayer.get_playback_position() + AudioServer.get_time_since_last_mix() - AudioServer.get_output_latency()

Now use 'current_music_time' to trigger your rhythm events

-
- Initial Playback Delay: When you call `play()` on an `AudioStreamPlayer`, the sound doesn't start immediately. You can estimate this initial delay by adding `AudioServer.get_time_to_next_mix()` and `AudioServer.get_output_latency()`. This is useful for timing the start of your game logic with the start of the music.

2. Building a "Conductor" Node

Many rhythm game tutorials advocate for a dedicated "Conductor" node. This node is responsible for:

- Storing Music Data: BPM, time signature, and potentially a "chart" of upcoming events (notes, enemy spawns, etc.) with their corresponding beat/time.
- Calculating Beat Times: Based on the BPM, the conductor calculates the precise time (in seconds) for each beat, measure, or any other rhythmic subdivision.
- Emitting Signals: As the `current_music_time` (calculated with latency compensation) passes a scheduled event's time, the conductor emits a signal. Other game objects (e.g., note spawners, animations, UI elements) can then connect to these signals and react accordingly.

Example Conductor Node Structure (GDScript):

GDScript

extends Node

@export var bpm: float = 120.0

@export var audio_stream_player: AudioStreamPlayer

@export var initial_offset: float = 0.0 # Time in seconds before the first beat

signal beat_triggered(beat_number)

signal measure_triggered(measure_number)

```
signal custom_event_triggered(event_id)


var _current_beat: int = -1

var _current_measure: int = -1

var _beat_interval: float

var _start_time: float = 0.0


func _ready():

    _beat_interval = 60.0 / bpm

    if audio_stream_player:

        audio_stream_player.play()

        # Account for initial audio playback delay

        _start_time = Time.get_ticks_usec() / 1_000_000.0 + AudioServer.get_time_to_next_mix() + AudioServer.get_output_latency()

    else:

        # If no AudioStreamPlayer, just start counting immediately (useful for debugging or non-audio sync)

        _start_time = Time.get_ticks_usec() / 1_000_000.0


func _process(delta):

    var elapsed_time = (Time.get_ticks_usec() / 1_000_000.0) - _start_time - AudioServer.get_output_latency()


    # If using an AudioStreamPlayer, ensure we're getting the most accurate time

    if audio_stream_player and audio_stream_player.is_playing():

        elapsed_time = audio_stream_player.get_playback_position() + AudioServer.get_time_since_last_mix() - AudioServer.get_output_latency()
```

```
var target_beat = floor((elapsed_time - initial_offset) / _beat_interval)

if target_beat > _current_beat:
    _current_beat = target_beat
    emit_signal("beat_triggered", _current_beat)
    # You can also emit measure_triggered here if you track time signature
    # For example, if 4 beats per measure:
    # var target_measure = floor(_current_beat / 4.0)
    # if target_measure > _current_measure:
    #     _current_measure = target_measure
    #     emit_signal("measure_triggered", _current_measure)

# Example of triggering custom events from a pre-defined chart
# This would involve loading an array of dictionaries or custom resources
# for your song's events:
# for event in song_chart:
#     if elapsed_time >= event.time_to_trigger and not event.triggered:
#         emit_signal("custom_event_triggered", event.id)
#         event.triggered = true # Mark as triggered to avoid multiple emissions
```

3. Using the `RhythmNotifier` Addon

Godot's Asset Library offers the `RhythmNotifier` addon (Godot 4). This addon provides a `RhythmNotifier` class that simplifies beat synchronization.

- **Features:** It emits rhythmic signals synchronized with an `AudioStreamPlayer`, accounting for output latency. You can define custom signals to emit on specific beats or repeatedly.
- **Usage:**
  1. Install `RhythmNotifier` from the Godot Asset Library.
  2. Add a `RhythmNotifier` node to your scene.
  3. Set its `BPM` property.
  4. Optionally, drag your `AudioStreamPlayer` node onto its `Audio Stream Player` property.
  5. Use its `beats()` method to create and connect to signals.

GDScript

```
# Example using RhythmNotifier (from its documentation)

@onready var rhythm_notifier: RhythmNotifier = $RhythmNotifier



func _ready():

   rhythm_notifier.bpm = 120.0

   rhythm_notifier.audio_stream_player = $MyAudioPlayer # Assign your AudioStreamPlayer

   rhythm_notifier.beats(1).connect(on_beat) # Emit signal every beat

   rhythm_notifier.beats(4, false, 3).connect(on_specific_beat) # Emit on beat 3 (once)

   $MyAudioPlayer.play()



func on_beat(beat_count: int):

   print("Beat! Current beat: ", beat_count)

   # Trigger visual effects, spawn notes, etc.



func on_specific_beat(_beat_data):

   print("Specific beat reached!")
```

-

This addon can significantly reduce the boilerplate code needed for rhythm synchronization.

4. Charting System (for complex rhythm games)

For games like Guitar Hero or Friday Night Funkin', you'll need a robust charting system:

- **External Data: Instead of hardcoding events, define your song's events (note timings, enemy spawns, etc.) in an external format (JSON, CSV, custom text file).**
- **Loading Charts: Your game loads this chart data at runtime.**
- **Event Queue: As the music plays, the conductor (or a dedicated chart reader) checks the chart for upcoming events based on the current music time and triggers them.**
- **Editor: For complex games, you might even build a simple in-game or external editor to create these charts visually, aligning events with the music.**

Key Considerations for Precision

- **Audio File Format: Use uncompressed formats like WAV for rhythm-critical audio to minimize decoding overhead and potential delays.**
- **Project Settings (Audio Latency): In Godot's Project Settings, under `Audio -> Driver -> Output Latency`, you can try reducing the default value (e.g., from 15ms to 5ms) to minimize audible delay. Be aware that lower latency can increase CPU usage and potentially lead to audio crackling if your system struggles.**
- **`_process` vs. `_physics_process`: For rhythm game logic, it's generally better to base your timing on `_process` (which runs every frame) or the audio thread's actual position rather than `_physics_process` (which has a fixed tick rate, potentially leading to less granular timing).**
- **User Calibration: For the best player experience, consider implementing an in-game calibration system where players can adjust a global offset to compensate for their specific audio setup and their own reaction time. This allows them to fine-tune the synchronization to feel perfect.**
- **Visual and Audio Cues: Ensure your visual cues (notes appearing, hit animations) and audio cues (hit sounds, music changes) are tightly synchronized. A slight visual offset can be tolerated more than an audible one.**

By combining these strategies and paying close attention to Godot's audio timing features, you can successfully implement robust rhythm synchronization in your projects.

To implement computer vision in a dance rhythm game using a camera, follow this structured approach:

1. Pose Detection
Technology: Use MediaPipe Pose or OpenPose for real-time body landmark detection.

Keypoints: Track critical joints (wrists, elbows, shoulders, hips, knees, ankles).

Output: 2D/3D coordinates of body joints from each camera frame.

2. Move Definition & Tracking
Predefined Moves: Create a library of dance moves (e.g., arm raises, squats, spins) with expected joint positions/angles.

Tracking Logic:

Calculate joint angles (e.g., elbow angle via vectors between shoulder-elbow-wrist).

Measure limb positions relative to the body (e.g., "left hand above head").

Detect motion vectors (e.g., rapid upward leg movement for a kick).

3. Rhythm Synchronization
Beat Mapping: Design beat maps (like DDR step charts) with:

Timestamps: When moves should be performed.

Move IDs: Expected move for each beat.

Timing Windows: Define hit windows (e.g., ±250ms around the beat) for scoring.

4. Move Validation & Scoring
Similarity Check: Compare player's pose with the expected move using:

Cosine similarity for joint angles.

Euclidean distance for keypoint positions.

Scoring:

Perfect/Good/Miss based on timing and pose accuracy.

Combo multipliers for consecutive hits.

5. Game Interface
Feedback Overlays:

Display the next move (e.g., arrow directions, limb icons).

Show real-time pose with skeleton overlays.

Highlight scoring (e.g., "PERFECT!") and combo counters.

Camera Feed: Show player’s video with augmented graphics.

6. Technical Implementation
Tools:

Python with OpenCV (camera handling).

MediaPipe (pose estimation).

PyGame or Unity (game UI/rendering).

LibROSA (audio beat tracking).

Workflow:

Capture camera frames.

Run pose estimation per frame.

For each beat in the map:

Check if current time is within the beat’s hit window.

Validate pose similarity to the expected move.

Update score and feedback.

Render UI with camera feed, move prompts, and score.

7. Optimization Tips
Robustness:

Use filtering (e.g., Kalman filter) to smooth jittery joint data.

Handle occlusions via interpolation or historical data.

Calibration: Normalize poses to the player's height/arm span at the start.

Performance: Reduce input lag by optimizing pose estimation (e.g., lower-resolution processing).

Example Code Snippet (MediaPipe + PyGame)
python
```
import cv2
import pygame
import mediapipe as mp

# Initialize MediaPipe Pose
mp_pose = mp.solutions.pose
pose = mp_pose.Pose()

# PyGame setup
pygame.init()
screen = pygame.display.set_mode((800, 600))

# Beat map example: [(time_ms, expected_move)]
beat_map = [(1000, "ARM_UP"), (2000, "SQUAT")]

def check_move(expected, landmarks):
    if expected == "ARM_UP":
        # Check if wrists are above shoulders
        left_wrist = landmarks[mp_pose.PoseLandmark.LEFT_WRIST]
        left_shoulder = landmarks[mp_pose.PoseLandmark.LEFT_SHOULDER]
        return left_wrist.y < left_shoulder.y
    elif expected == "SQUAT":
        # Check if hips are below knees
        left_hip = landmarks[mp_pose.PoseLandmark.LEFT_HIP]
```

```
        left_knee = landmarks[mp_pose.PoseLandmark.LEFT_KNEE]
        return left_hip.y > left_knee.y
    return False

cap = cv2.VideoCapture(0)
clock = pygame.time.Clock()
start_time = pygame.time.get_ticks()

while True:
    # Process camera frame
    ret, frame = cap.read()
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = pose.process(rgb_frame)

    # Get current time
    current_time = pygame.time.get_ticks() - start_time

    # Check beat map
    for beat_time, move in beat_map:
        if abs(current_time - beat_time) < 250:  # 250ms window
            if results.pose_landmarks and check_move(move,
results.pose_landmarks.landmark):
                print("HIT!")

    # Render UI (simplified)
    screen.fill((0, 0, 0))
    # ... draw game elements here ...
    pygame.display.flip()
    clock.tick(30)
```

Challenges & Solutions

Latency: Use lightweight models and skip frames if needed.

Varying Body Types: Normalize joint positions to the player’s torso size.

Complex Moves: Break moves into phases (e.g., "wind-up" and "kick").

Lighting/Background: Use background subtraction or depth cameras.

This approach creates an engaging, movement-based rhythm game where players physically perform dance moves tracked via camera.

Implementing rhythm synchronization in Godot involves aligning game events (like character movements, enemy attacks, or visual effects) to the beat of background music. Here's a step-by-step guide:

1. Core Concept: The Conductor
Create a "Conductor" (autoload/singleton) that tracks musical timing. It calculates beat positions using the song's BPM and compensates for audio latency.

Conductor Script (conductor.gd):

```
gdscript
extends Node

# Configuration
var bpm: float = 128.0  # Beats Per Minute
var measures: int = 4    # Beats per measure (e.g., 4/4 time)
var song_position: float = 0.0
var song_position_in_beats: int = 0
var sec_per_beat: float = 0.0
var last_beat: int = 0
var offset: float = 0.45  # Calibration offset (adjust per project)

# Audio
var music_player: AudioStreamPlayer

# Signals
signal beat(position)     # Emitted every beat (position = beat count)
signal measure(position)  # Emitted every measure (position = measure count)

func _ready():
    sec_per_beat = 60.0 / bpm
    music_player = $AudioStreamPlayer  # Attach your AudioStreamPlayer node

func start_song():
    song_position = 0.0
    song_position_in_beats = 0
    last_beat = 0
    music_player.play()

func _process(_delta):
```

```
    if music_player.playing:
        # Calculate position with latency compensation
        song_position = (
            music_player.get_playback_position() +
            AudioServer.get_time_since_last_mix() -
            AudioServer.get_output_latency()
        ) - offset

        # Track beats
        song_position_in_beats = int(floor(song_position / sec_per_beat))

        # Emit signals on new beats
        if song_position_in_beats > last_beat:
            last_beat = song_position_in_beats
            emit_signal("beat", song_position_in_beats)

            # Emit measure signal every X beats
            if song_position_in_beats % measures == 0:
                emit_signal("measure", song_position_in_beats / measures)
```
2. Setup the Conductor
Create the Conductor:

Attach conductor.gd to a node in a new scene.

Add an AudioStreamPlayer as its child (load your music file).

Save the scene as Conductor.tscn.

Register as Autoload:

Go to Project > Project Settings > Autoload.

Add Conductor.tscn as a singleton.

3. Synchronize Game Elements
Connect game objects to the Conductor's signals to trigger timed actions.

Example: Character Jump on Beat

gdscript

```
extends CharacterBody2D

func _ready():
    Conductor.connect("beat", _on_beat)

func _on_beat(beat_count):
    if beat_count % 2 == 0:  # Jump every 2 beats
        velocity.y = -500
        jump_animation.play()
```
Example: Enemy Attack on Measure

```gdscript
extends Node2D

func _ready():
    Conductor.connect("measure", _on_measure)

func _on_measure(measure_count):
    if measure_count % 2 == 0:  # Attack every 2 measures
        start_attack_pattern()
```
4. Calibration & Testing

Offset Calibration:

Adjust offset in the Conductor until actions align perfectly with beats.

Use a debug label to show beat numbers:

```gdscript
func _on_beat(beat_count):
    $DebugLabel.text = "BEAT: %d" % beat_count
```
Visual Beat Indicator (Optional):

Create a UI element that flashes on beat:

```gdscript
func _on_beat(_beat):
    $BeatIndicator.modulate = Color.RED
    await get_tree().create_timer(0.05).timeout
    $BeatIndicator.modulate = Color.WHITE
```
5. Advanced Techniques

Sub-Beats: Divide beats for 8th/16th notes:

gdscript
```
var sub_beats = 4  # Quarter notes per beat
var current_sub_beat = int(floor(song_position / (sec_per_beat / sub_beats)))
```
Dynamic BPM: Modify bpm/sec_per_beat during runtime for tempo changes.

Input Timing Windows:

gdscript
```
func _input(event):
    if event.is_action_pressed("action"):
        var time_since_beat = fmod(song_position, sec_per_beat)
        if time_since_beat < 0.1:  # 100ms window
            perfect_hit()
```
Troubleshooting Tips
Audio Latency: Test on target devices; adjust offset per platform.

Missed First Beat: Start music slightly delayed using a Timer.

Drift Issues: Use AudioStreamPlayer (not AudioStreamPlayer2D/3D).

This setup creates a robust rhythm synchronization system. Adjust BPM, offset, and signal connections to match your game's requirements!