TiltGolf
Team Name: TiltGolf

Noah Hathout {nhathout}, Bennett Taylor {betaylor}, Skanda Nadig {skandan}

Github Repo: https://github.com/nhathout/TiltGolf
Demo Video YouTube Link: https://www.youtube.com/watch?v=vEOcKTJfxNo

Figure 1. TiltGolf handheld gaming console.

Abstract:
TiltGolf is a tilt-controlled mini-golf game that runs on the BeagleBone Black development board with an attached LCD cape. An IMU measures tilt and reads those values into the Qt game, which are then used as user input for moving the ball. The tilt readings are calibrated and smoothed to provide responsive and accurate input. The hardware is all contained within a 3D printed enclosure to create the overall hand held device. The UI presents arcade-style menus and levels, while the physics layer keeps the golf ball rolling realistically around the different levels and obstacles. There are 6 levels of increasing difficulty for players to complete, with walls and water hazards acting as obstacles in the journey to the hole.

1. Introduction

TiltGolf is a tilt-controlled, physics-based golf game implemented on a BeagleBone Black [1]. The overall system turns the BeagleBone into a functional handheld gaming device where players use the touchscreen to navigate between levels and modes, and tilt the device itself to move a golf ball through a level and into the target hole (indicated with a small red flag). To make this more engaging, we have included six fully custom levels which increase in difficulty and unlock progressively.

The motivation for this project came from our collective desire to recreate the feeling of playing simple yet addictive retro games, while maintaining the quality and interactivity of modern day handheld gaming devices. When thinking of games we grew up with, the tilt sensor functionality of the early Wii controllers [2] came to mind. Combining these ideas led us to a design without conventional buttons or D-pads. Instead, TiltGolf uses a touchscreen and an Inertial Measurement Unit (IMU) [3] to act as a tilt-controller, which is what controls the ball movement.

At a physical-level, the project consists of four major components. In terms of compute, the BeagleBone Black development board was utilized for processing game and display logic. For displaying the game to the user and gathering touch input, a 4.3" LCD Capacitive Touchscreen Display Cape [4] provided the perfect platform. A fully 3D-designed and printed case protects the internal hardware and turns the embedded system into a handheld console. Finally, the IMU communicates to the system via file descriptors and the I²C communication protocol [5]. It has its own dedicated calibration algorithm to limit drift and noise.

On the software-level, the Qt application framework and Box2D physics engine provided the core structure for our game. QT visualization layer implements the menu screen, level selection, the free-play mode, and all level-specific game visualization. Another control layer in QT takes care of the level transitions, reset conditions, and the level-unlocking mechanism that unlocks levels as players complete them in order. We developed the physics layer using Box2D [6] to simulate the physics for the golf ball, the walls (which we purposely made bouncy), water hazards, and the target hole.

In TiltGolf currently, we accomplished a fully functional handheld device that features an intuitive touchscreen menu which refreshes at 30FPS, accompanied by a calibratable tilt-based control system which measures tilt at 20 Hz. We maintain enough compute to fully handle the physics simulation, level visualization, and navigation system without any undesired effects like lag or clipping. Ultimately we accomplished a simple yet fun main menu layout, free-play menu, and a multi-level campaign where levels unlock sequentially and get increasingly more difficult.

2. Design Flow

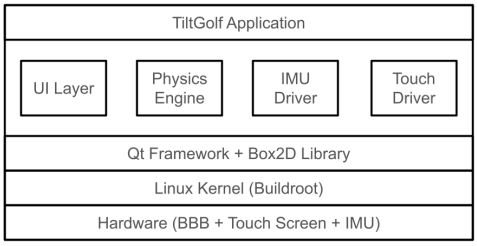

Figure 2. High-level system architecture

System Components

1. Platform: Qt5 UI + Box2D physics, cross-built for an ARM based embedded target running Linux
2. Hardware I/O: Beaglebone Black, LCD touchscreen, IMU sensor(I2C comm).
3. Runtime loop: GameController drives a QTimer game loop (~60 Hz) that advances PhysicsEngine, reads IMU, and triggers view and repaints the screen.
4. Rendering: GameView paints the whole frame (opaque) every update to avoid artifacts.

5. Levels: LevelData.h supplies level geometry and special features (level advance, water hazards, walls, ball start, goal/hole).
6. IMU Calibration: Non-persistent in-memory calibration and preview via CalibrationDialog.

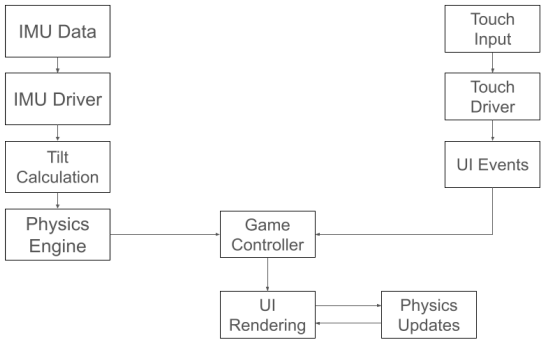

Figure 3. Data flow chart

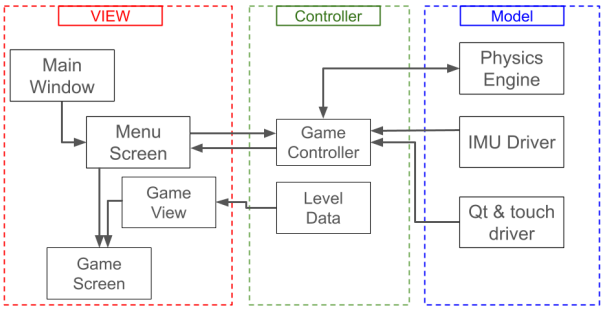

Figure 4. Control Flow

Data and Control flow (user & runtime)

1. Startup
    a. main.cpp → MainWindow constructs MenuScreen and GameScreen.
2. Level selection and progress
    a. User clicks MenuScreen button → emits levelSelected(id) → MainWindow::startLevel → GameScreen::setLevel → GameController::loadLevel
    b. GameController calls LevelData::getLevel(id) → PhysicsEngine::loadLevel builds Box2D world, walls, water sensors, and ball.
    c. Level unlocking as the player wins and advances in the game.
3. Game Control
    a. QTimer→GameController::gameLoop:
        i. physics→step() → IMU::update() → sensor mapping → ApplyForceToCenter → world→Step()
        ii. ContactListener updates ballInWater flag via sensor fixtures. PhysicsEngine exposes wasBallInWater().
        iii. GameController checks wasBallInWater() → emit gameOver; checks hole proximity → emit gameWon.
        iv. emit gameStateUpdated() → GameView receives and repaints the whole frame.
4. IMU Calibration
    a. User taps Calibrate → GameController::startCalibrationPreview(false) → PhysicsEngine::startCalibrationPreview → IMU::setTempBiasFromCurrentReading() (temp_bias applied immediately).
    b. CalibrationDialog shown non-modally; Save → acceptCalibrationPreview() → PhysicsEngine::commitCalibrationPreview() → IMU::commitTempBiasToSaved(); Cancel → cancelCalibrationPreview() → IMU::clearTempBias()

For project work split up, Bennett worked on the menuing and general QT architecture (screens and views) along with set up of general game visualization, physics and logic. Noah helped with initial physics set up, worked on the 3D printed enclosure, level design/data, finishing touches (level lock, win screen, etc.), and README. Skanda fully implemented the IMU initialization, communication, and calibration while also helping refine the game visualization and physics. The rough estimation of overall contribution goes as follows: Bennett Taylor 33%, Skanda Nadig 33%, and Noah Hathout 34%

3. Project Details

3.1 Visuals and Game UI

The TiltGolf UI is implemented in Qt and structured around a simple screen stack. MainWindow hosts a QStackedWidget that switches between MenuScreen and GameScreen. Application logic and state are separated from rendering. GameController drives the game loop, PhysicsEngine owns Box2D and IMU interaction, and GameView is the dedicated paint surface. This separation keeps UI code small and focused on layout and user interactions while delegating physics, sensor processing, and level geometry to the controller/engine layers.

The MenuScreen presents a full-screen, touch-friendly level selector and exit control. A vertical layout contains a prominent title and a 3×2 grid of large level buttons (six levels), with an Exit button in the top bar. Buttons emit a 1-based levelSelected(int) signal so UI labels match LevelData IDs. GameScreen composes the interactive game UI: a compact top bar (level label, elapsed time, Calibrate, Restart, Exit) and the large central GameView that renders the playfield. Top bar controls are connected to GameController actions (restart, pause/exit, calibration preview).

Rendering and painting loop GameView handles all visual rendering using QPainter and is configured as opaque (WA_OpaquePaintEvent) to paint the entire widget every frame. Each paintEvent clears the background to a green ground, then draws water (light fill + dark edge), brown walls, the hole/flag, start marker, and ball in that back-to-front order. Repaints are driven by GameController::gameStateUpdated emitted from the ~60 Hz QTimer loop; the view simply calls update() to schedule paintEvent, keeping paint code deterministic and optimized for the Beaglebone's CPU.

Touch, calibration and interaction behavior Touch interactions are standard QPushButton taps; the CalibrationDialog is a compact, non-modal touch dialog placed bottom-center so the user can tilt the device while previewing. Calibration uses an in-memory saved bias plus a temporary preview bias so the current pose can be tested without committing; Save commits the preview, Cancel clears it. The UI avoids long blocking operations on the main thread (except short calibration averaging when enabled), and the project uses deadzone + smoothing in the physics path to make motion feel less jittery for touch and tilt inputs.

3.2 Game Physics

a. Input source: IMU (magnetometer) values returned by IMU::getX()/getY(). PhysicsEngine::step() reads these values each tick via imu.update().
b. Axis mapping: raw sensor axes are remapped depending on mounting:
    i. swapXY, invertX, invertY flags let you correct orientation without changing user code.
c. Scaling: raw sensor values are converted into world forces by multiplying by a sensitivity constant k_Force (PhysicsEngine.h).
    i. raw_fx = sensorX * k_Force
    ii. raw_fy = sensorY * k_Force
d. Application of force: the computed force is applied to the ball with ballBody->ApplyForceToCenter(b2Vec2(fx, fy), true) inside PhysicsEngine::step().

Smoothing and jitter reduction

a. Deadzone: tiny raw force values are suppressed using a deadzone threshold:
    i. if (fabs(raw_fx) < deadzone) raw_fx = 0.0f; same for raw_fy.
    ii. Ignore sensor noise when the device is near neutral, preventing micro-forces that cause jitter.

b. Linear & angular damping:
    i. When creating the ball body, linearDamping and angularDamping are set (ballDef.linearDamping = 1.0f; ballDef.angularDamping = 0.8f).
    ii. Damping passively reduces velocities over time, preventing perpetual small oscillations and making motion feel "natural" and stable.

3.3 Game Controller

3.3.1 Design and Purpose

GameController coordinates game state, inputs, and high-level rules. In TiltGolf the GameController owns the PhysicsEngine, starts and stops the game loop timer, exposes calibration and reset actions to the UI, and emits signals (gameStateUpdated, gameWon, gameOver) that the view layer listens to.

3.3.2 Level loading and reset logic

To start a level the controller pulls LevelConfig from LevelData, instructs the PhysicsEngine to build the Box2D world (walls, water sensors, ball spawn), resets internal flags and smoothing state, and starts the QTimer that drives the simulation (~60 Hz). Resetting a level is a deterministic controller action: it tells PhysicsEngine to move the ball to the start pose, zero velocities, clear transient state, and resume the timer so the game restarts cleanly without UI code needing to know physics implementation details.

3.3.3 Gaming Logic

The controller implements the core game loop and win/hazard detection. On each QTimer tick GameController calls physics->step() (which reads the IMU, computes and smooths forces, steps Box2D, and updates contact flags). After stepping, the controller evaluates win and loss/restart conditions

3.3.4 Screen transitions and UI interactions

The Controller mediates UI/UX through signals and simple public methods. MenuScreen emits levelSelected(id) which MainWindow forwards to the controller (via GameScreen::setLevel→GameController::loadLevel). The GameScreen calls controller methods for restart, calibration preview/commit/cancel and pauses the controller before switching back to the menu to avoid background simulation. This signal/slot interaction allows UI widgets to request actions and subscribe to high-level state changes from the controller.

3.3.5 Level locking

We implement a level locking/unlocking mechanism as a small state machine in MainWindow. On startup, MainWindow gets a std::vector<bool> unlockedLevels with an entry per level (6 for now). The base case is that only level 1 holds true. Tapping the "Free Play" button turns on a freePlayMode flag that refreshes the grid so that every level is unlocked. Level unlocks are achieved from the gameWon signal emitted by GameController.

3.4 Physical Interfaces: Touchscreen and IMU

3.4.1 Touchscreen:

The BeagleBone display cape provides a fully integrated solution for video output and touch input. The LCD uses the AM335x's parallel RGB interface through cape-to-panel FPC connectors, while the touchscreen communicates via I²C using standard Linux drivers. Qt's input framework makes the touchscreen transparent to the application, enabling a smooth user experience for the TiltGolf game. In the GameScreen class, all user interaction—such as pressing Restart, Exit, and Calibrate, or interacting with dialogs—is handled through Qt widgets (QPushButton, QMessageBox, custom calibration dialogs), which automatically receive touch input the same way they would receive mouse clicks. Qt converts

screen touches into QMouseEvent or QTouchEvent objects, and the game's UI elements connect directly to their corresponding signals (clicked, etc.) without needing to manually read low-level I²C or /dev/input events. This abstraction allows the touchscreen to operate as a fully integrated GUI input device within the game while maintaining a clean separation between touch handling and gameplay logic.

3.4.2 IMU (Polulu MiniIMU-9 v5 [3])

The MiniIMU-9 is a system-in-package featuring an LSM6DS33 3-axis gyro and an LIS3MDL 3-axis magnetometer.

IMU setup

1. The IMU has 5pins with the following description:

Pin#	Name	Function
1	SCL	Signal interface I2C serial clock (SCL)
2	SDA	Signal interface I2C serial data (SDA)
3	GND	0V supply (Ground)
4	VIN	Power Supply (3.3V Input)
5	VDD	Power Supply (Regulated Voltage)

Table 1: Pin Description

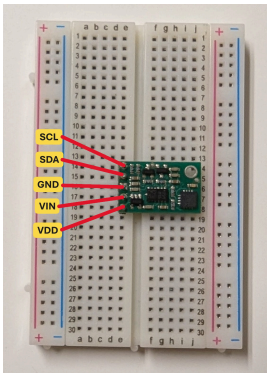

Figure 5: Pin outs on IMU

2. The IMU is connected to the Beaglebone over the I2C bus. Pull up resistors to SCL and SDA pins are required to support standard I2C. The connections are shown in the circuit diagram below.

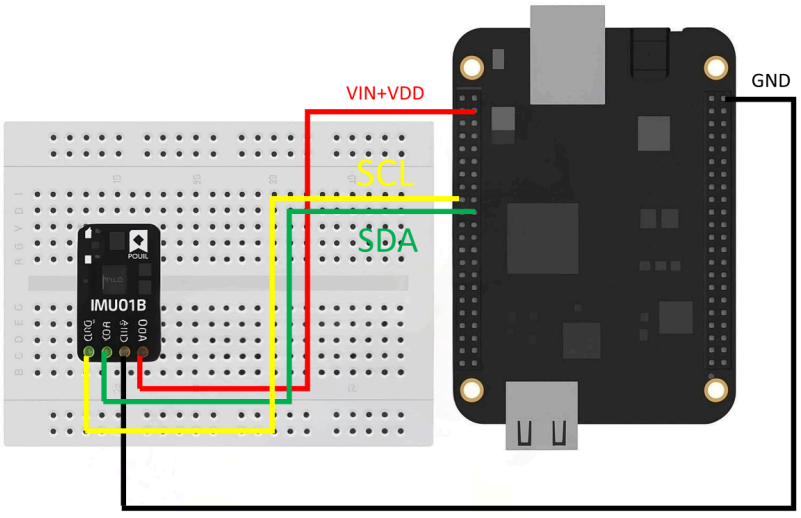

Figure: Circuit diagram (IMU connections)

3. Read/Write from IMU

   Register values of the IMU can be accessed over I2C. Implementations in the code:

   i. I2C device path: I2C_DEVICE = "/dev/i2c-2"
   ii. Magnetometer I2C address: MAG_ADDR = 0x1E
   iii. open(I2C_DEVICE, O_RDWR)
   iv. ioctl(i2c_fd, I2C_SLAVE, MAG_ADDR): Select the magnetometer slave address for subsequent I2C transactions. If ioctl fails, the function closes the fd and returns false.
   v. Configure magnetometer registers by writing bytes to set data output rate and averaging, gain, and mode
   vi. Clear saved and temporary biases (bias_x/y/z and temp_bias_x/y/z = 0)
   vii. Return true on success.

IMU optimizations

1. Sensor Data Rate:
   a. Magnetometer: configurable ODR via CRA_REG_M.
   b. I²C speed is set by the hardware and device tree.
   c. Realistic Update Rate for Game
      i. Magnetometer: 20 Hz → new data every ~88 ms
      ii. If UI updates every 200 ms (like in Qt timer), sensor update rate is not exceeded.
2. Smooth movement game physics
   a. A simple low-pass smoothing is applied in the physics step so the force applied to the ball is smoothened.
   b. The filter is a first-order Infinite Impulse Response (IIR) low-pass:
   $f_x[n] = f_x[n-1] + \alpha \cdot (f_{x_{raw}}[n] - f_x[n-1])$

3.4.3 3D Printed Enclosure

As mentioned before, the TiltGolf hardware is contained within a fully custom 3D-printed enclosure that gives this project its retro feel while also ensuring the whole system can be handheld. Our first housing (v1), shown on the left in Figure 6, was taken directly from an open source enclosure design from Thingiverse [7]. That model was a good starting point for the general press-fit connections, the LCD hole on the front cover, and the side holes for the cable connections to the BeagleBone.

Once we began laying out the actual hardware, it was obvious that some changes were needed. Firstly, the initial version included buttons, which our project did not need. Next, the internal volume was not enough to place all of our hardware, therefore we increased the length, width, and height of the entire enclosure accordingly. We also realized that the BeagleBone needed to be mounted upside down on stilts/standoffs so that the LCD cape could reach the front opening cleanly. This then required us to higher the cutouts for the cable connectors (such as the power cable) so that everything aligned properly. The full v2 enclosure is shown on the right in Figure 6 (see /assets/housing/v2/bbbholder.stl to view or download the BeagleBone mount model).

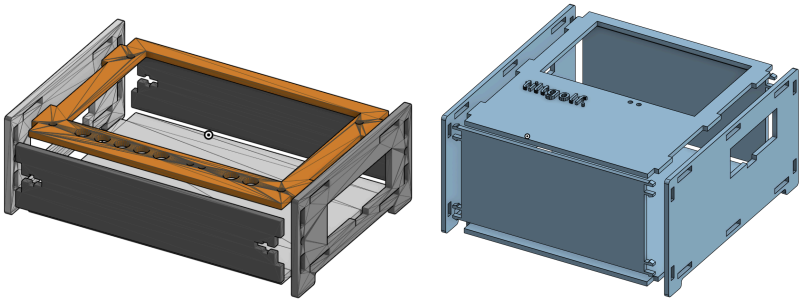

Figure 6. Full custom CAD assembly of 3D-printed enclosure. Left: initial, Right: final

4. Summary

The TiltGolf project delivers a complete, functional hand-held gaming device and corresponding application which together provides a user with a fun tilt-controlled minigolf style game. The game currently has 6 levels of progressively increasing difficulty with a start, end (the hole), and obstacles in between. Users must navigate walls and water hazards by tilting the device to "roll" the ball in a desired direction.

The physical device consists of a custom 3D printed enclosure, a BeagleBone Black development board, an IMU, and a 4.3" touchscreen LCD display. In software the QT framework was utilized for application creation, menuing, game control logic, and general visualization. For the physics simulation of the ball, walls and hazards, the Box2D library was used to provide accurate and fast collision and position data without major errors.

References

[1] BeagleBone.org Foundation. "BeagleBone® Black". https://www.beagleboard.org/boards/beaglebone-black
[2] Wikipedia. "Wii Remote". https://en.wikipedia.org/wiki/Wii_Remote
[3] Polulu. "MinIMU-9 v5 Gyro, Accelerometer, and Compass (LSM6DS33 and LIS3MDL Carrier)". https://www.pololu.com/product/2738
[4] 4.3" LCD Capacitive Touchscreen Display Cape for BeagleBone. https://www.adafruit.com/product/3396?srsltid=AfmBOoqBWQFD1gEHskTOMd_GiCcYx_6oscPc1etdkTzmjiBHjSh3WHPT
[5] I2C. https://learn.sparkfun.com/tutorials/i2c/all
[6] Box2D Documentation. https://box2d.org/documentation/
[7] BeagleBone Black with LCD Enclosure. https://www.thingiverse.com/thing:550532