Niels Joubert, Rohit Nambiar, Navin Lal, and Mark Sandstrom

Rockband Vision is a computer vision system capable of playing the Rockband console game autonomously. Rockband is an interactive band simulator that allows human players to play music according to on-screen musical notation using custom musical instruments. Our system is capable of interpreting the Rockband video feed presented to human players: it is able to read the musical notation in the feed and manipulate the game's guitar input to play the game. This project relied on a breadth of knowledge from the computer graphics, numerical analysis and physical computing areas. We present it as our final project for Berkeley's Computer Graphics course. In this paper we will present our design and implementation of this system, including the challenges encountered and design decisions made while building the system.

Problem Analysis

Rockband present the player with a guitar fretboard, rendered from the perspective of someone looking along the guitar's neck. Along the fretboard are 5 strings — 5 regions where notes can be drawn — lying along the horizontal axis. The vertical axis of the fretboard represents time, and notes move along it accordingly. At the bottom of the fretboard is the region that defines the current state of the guitar. If a note enters this region, the player is expected to push the strum toggle while simultaneously hitting the corresponding note button on the guitar. Notes move vertically along the 5 strings from the top of the fretboard to this region.

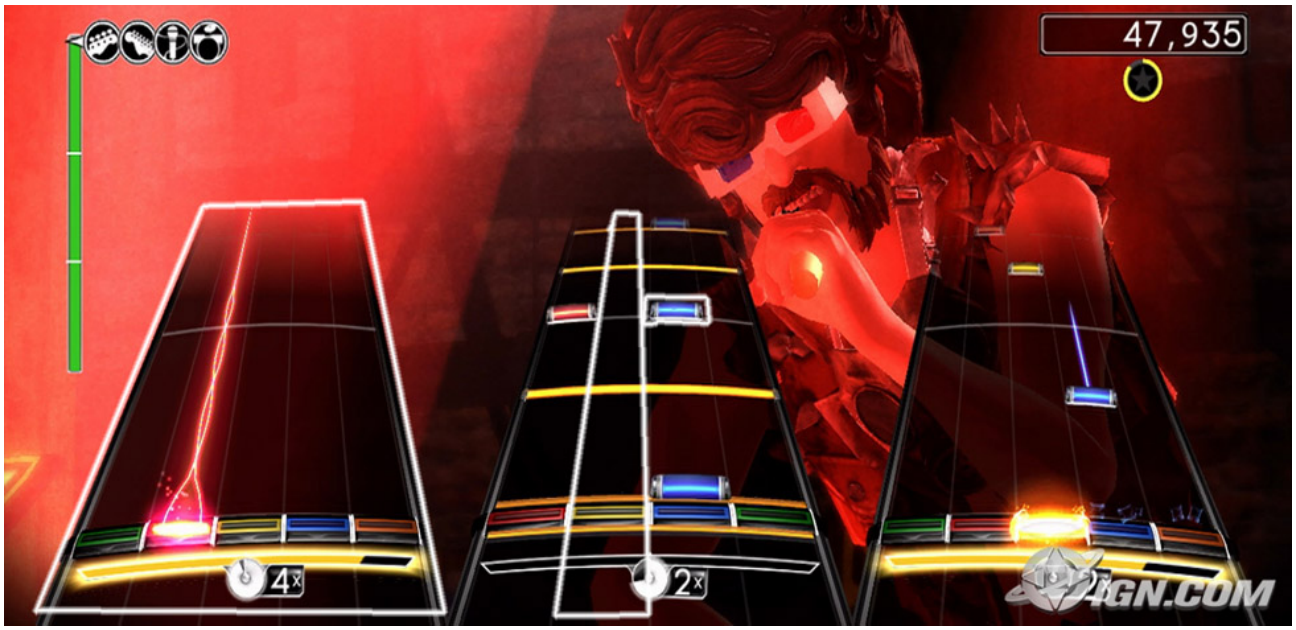


Figure: The Rockband screen, with white highlighted regions (left to right): Fretboard, Single String and Note

Our computer vision system monitors the fretboard presented to the player with the aim of imitating the guitar's state as notes move into the region where a player would play the note on the guitar. The fretboard stays a consistent shape, but the relative size depends on the amount of people playing a multiplayer game. The absolute size of this region depends on several other factors including camera model and physical setup. The fretboard is semitransparent black, allowing animations of bands to be seen through it. It can also display various patterns as players gain energy. Lines signifying the barlines in musical notation moves across it at speeds relative to the tempo of the song.

Notes move across the board at a constant speed throughout a song, but tempo differences between songs are manifested as variations in the speed of notes along the fretboard. Notes are rectangular blocks of various colors, and the colors can change as different scenarios unfold during gameplay. Notes can have tails in the form of colored lines following the note. Normal notes are expected to be played with a single hit of the strum toggle switch, while notes with tails need to be played by keeping the strum toggle engaged for the duration of the line. During this time the color and brightness of the tail changes.

Proposed Solution

We segmented the project into 3 major sections to consider:

- video capturing and processing
- note recognition, timing and tracking
- hardware interfacing

We need real-time pixel-level access to a video feed coming from a camera pointed at the screen. The project becomes extremely simple if you have access to the actual video feed from the XBox (as done by the "Slashbot" and "AutoGuitarHero" projects) but we specifically wanted to work with the video feed coming from a camera, mimicking the visual process our brain needs to work with. To capture and process video, we used the OpenCV library. This gave us easy access to the pixel data of the video feed and provided excellent matrix libraries.

We want to have information about the state of each string to detect and track notes as they move across the string. To achieve this we want to inverse the perspective transform applied by Rockband to present the moving notes, and divide the fretboard into 5 regions. This is our first major challenge - given a matrix of pixels, how do you find a fretboard? Edge detection and feature detection techniques might automate this step in the future, but we decided to skirt this challenge by asking the user to define 4 points on the corners of the fretboard for us. This makes the perspective transform and segmentation steps trivial.

We now need to find notes on each string. Since note colors can change, we decided to work with the luminance over each string. We convert the 5 strings - 5 rectangular regions on which notes appear - into 5 arrays of "brightness" values. We can find notes by searching the luminance array for the characteristic features of a note's luminance on the string. To do this, we build a template note and convolve it with the luminance array. This has the effect of accentuating the notes matching the template in a recognizable manner. We can now apply derivative analysis with specific parameters to find the peaks created by notes on the string.

Our design so far produces a list of positions in a frame. We now need to track these positions over multiple frames, finding the point in time when they need to translate into a button press on the guitar. We drew up several designs for tracking notes. Initially we considered finding notes as they move across a first region, time them until they reach a second region, and use this data to compute note velocity, allowing us to predict when it would reach the point where a hardware event has to occur.

We discarded this idea for a two-stage approach that is more resilient to noise. Each frame's luminance plot gets added and averaged to a common buffer for each string. We approximate how far each note moved between the previous and current frame to find how far to shift the buffer containing the old frames. We now monitor the lowest part of this buffer, sweeping a cursor over the region we're shifting each frame to find peaks to translate to pushing the guitar's buttons. This approach is both resilient to transient noises and accentuates the features we want to detect.

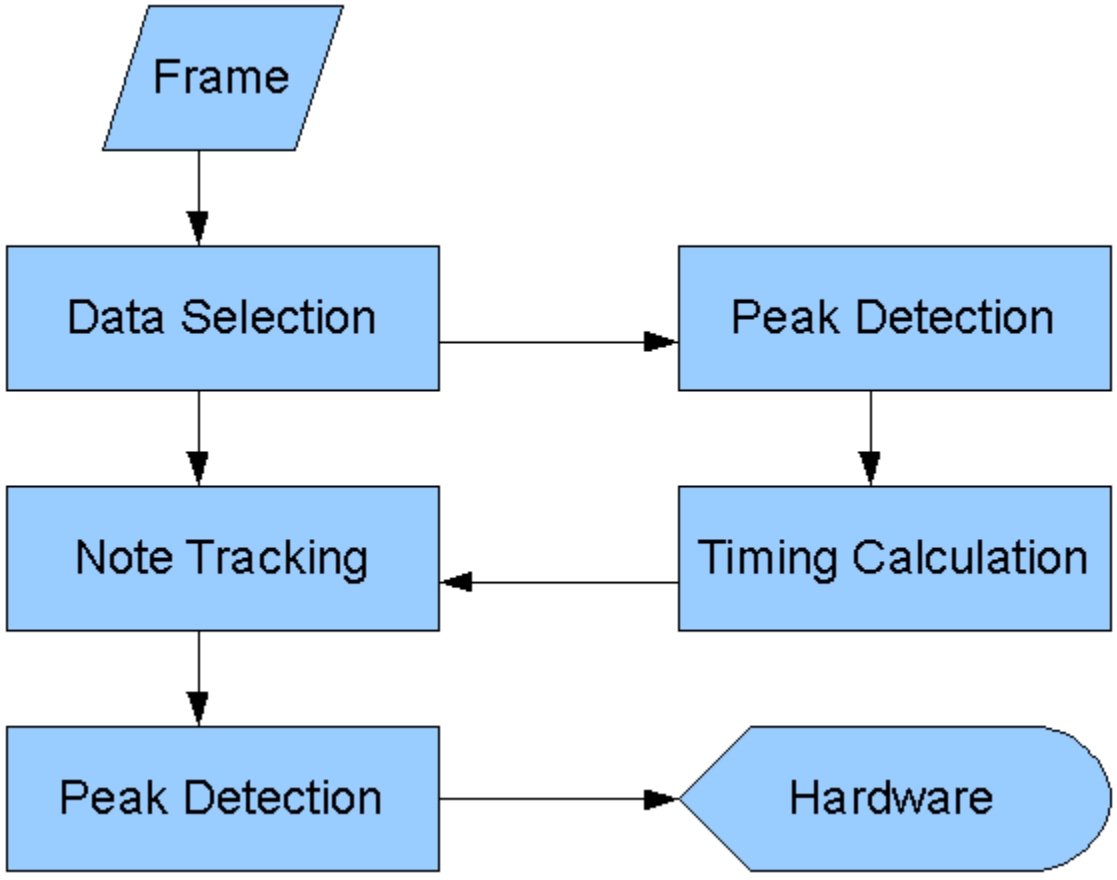
Design Analysis

This project requires a camera, computer, guitar controller, gaming console, and a display unit. The specifications of these items in our setup are shown below.

Camera: Panasonic Prosumer NV-DX100
Computer: 2.2Ghz Macbook Pro [2GB Ram]
Controller: Modified XBox Fender Statocaster controller
Gaming Console: XBox 360 with Rockband game
Display: HP DLP Projector, Samsung 20" Widescreen display (1:3000 contrast ratio)

A firewire cable was used to stream data from the camera to the PC, allowing for low-latency transmission. The guitar was modified with a prototyping board to allow for communication between the computer and the Xbox.

Our system can be described by the following flow diagram. A full explanation of each segment follows.



Data Selection

First we capture a frame from the camera. As a calibration step, the user must set up the four corners of a trapezoid identifying the area of the fretboard. While it would be possible to automatically detect this area, we decided to simplify our system by having the user manually specify reference points by clicking on the screen. An inverse perspective transformation is then calculated using these reference points. The transformation is applied to the captured frame, and the resulting image is cropped to the interior of the convex hull of the reference points, which is a rectangle after transformation. The result is that the notes travel at a constant velocity along the vertical access and remain at a fixed horizontal position.

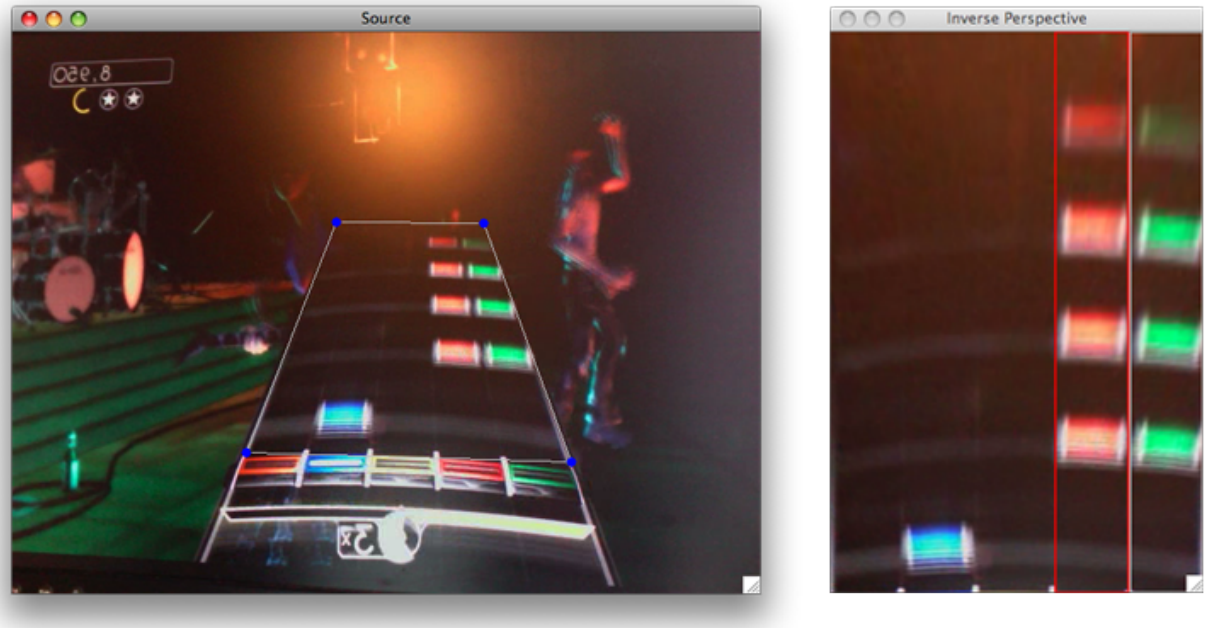


Figure: Source video and interface for specifying transformation reference points (left) and inverse perspective image (right).

We then convert the image to grayscale for further processing. This conversion is done by selecting the maximum RGB component for each pixel. Conversion in this manner allows for easier identification of notes since the notes are drawn using bright, saturated colors. No specific color information is actually used to identify the individual notes, meaning that the camera doesn't need to be color calibrated.

We then process each string. This is done by dividing the inverse perspective image into 5 equal-width images, one for each different type of note (green, red, yellow, blue, orange). A luminance plot is then generated for each string by averaging the pixels along each row of the individual string images. Notes are detected by identifying peaks in these luminance plots.

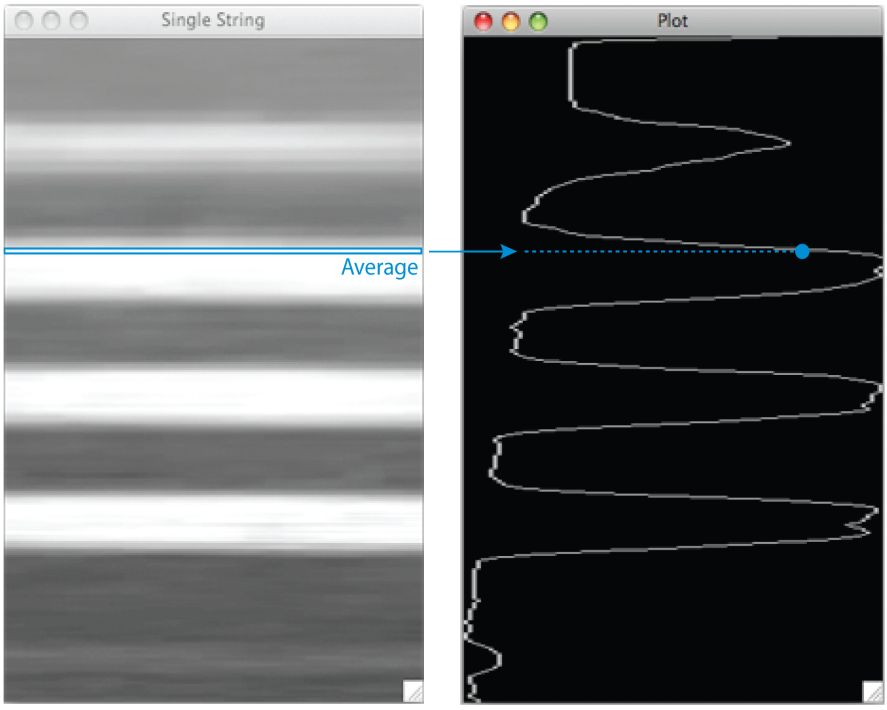


Figure: Gray scale image of a single string (left) and the corresponding luminance plot (right).

Peak Detection

Peak detection is the process by which the location of notes are pinpointed in the luminance data generated for each string displayed on the fretboard drawn by Rockband. The initial plot generated by the data selection module contain flat, noisy peaks where bright notes appear on the screen. To ease the process of peak detection, we filter this data with the aim of generating sharp peaks at the center of notes.

Convoluting the incoming data with a template of the note generates peaks where notes appears. This accentuates the position of the note and eases our peak detection. Since notes have a well-defined shape, we can estimate the correct note size for the template from the trapezoid defining the fretboard on screen. This is the same trapezoid used to generate the inverse perspective transform to extract the fretboard from the overall screen. The relation was empirically determined to be:

$$\text{average width} = (URx + LRx)/2 - (ULx + LLx)/2$$
$$\text{average height} = (LLy + LRy)/2 - (ULy + URy)/2$$
$$\text{template width } W = (\text{average width}) / (\text{average height}) * 40$$

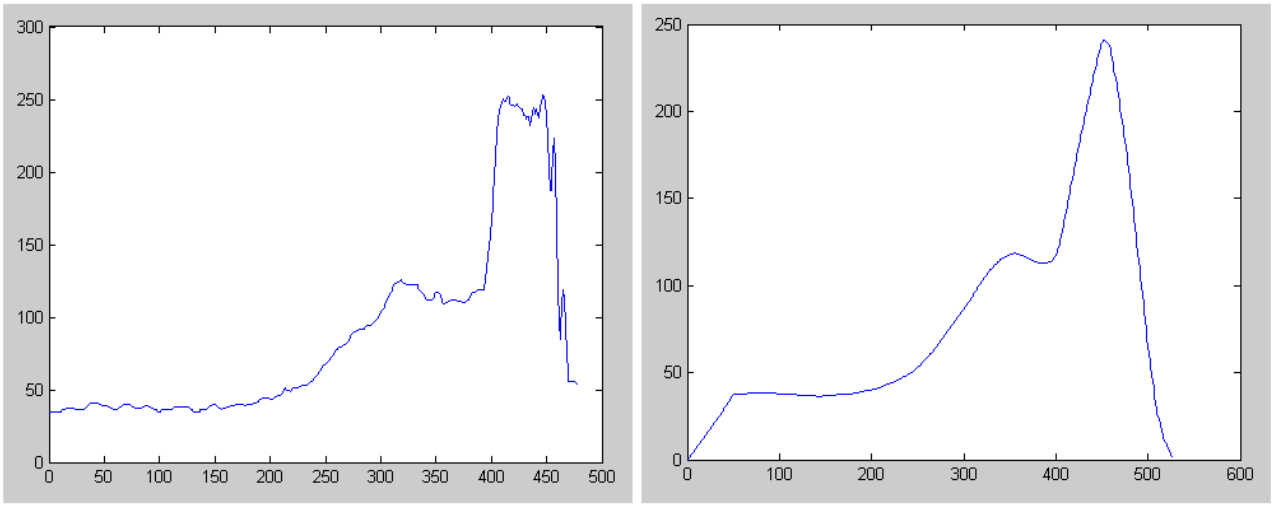


Figure: Example of raw data (left) and filtered data (convolution with 50-point box)

The data is thresholded to remove peaks in the median noise and retain only the bright peaks created by notes. The algorithm determining this threshold is an open-ended design question, and we attempted several approaches to dymanically adjust the threshold.

We decided to use the luminance data itself as the input to the thresholding algorithm. We found that the background brightness varies from string to string due to both background animations and outside interference from light sources affecting the camera and display. Thus each string has its own thresholding value, computed from the luminance of the string's region itself.

Since we're interested in the range of values that form the bright peaks in the data, we decided to calculate the threshold as a fraction of the standard deviation above the mean of the data. The fraction (alpha in the equation below) of standard deviation added to the mean gave us a parameter to tune this algorithm by. We initialized the algorithm with an alpha value 1.5, which worked satisfactorily.

$$\text{new threshold} = \text{mean} + \text{alpha} * \text{stddev}$$
$$\text{alpha} = 1.5$$

We want to create a thresholding algorithm robust enough to withstand the flashes in rockband's animation as well as short bursts of light due to changes in the physical environment around our camera and screen. We used exponential smoothing over time to create a temporally robust algorithm. By setting the threshold of any frame as a weighted sum of the previous threshold value and the newly calculated threshold, we refrain from making drastic changes to our thresholding value. The weight of the old to the new threshold gives us the second parameter by which to fine-tune this algorithm. For an initial value, we chose beta to be 19.0/20.0, weighing the current value much more than the newly calculated value.

$$\text{final threshold} = (1.0 - \text{beta}) * (\text{new threshold}) + (\text{beta}) * (\text{old threshold})$$

A drawback of this algorithm is the change in threshold that occurs as more or fewer peaks appear on screen. For a heavily populated string, the total bright area is significantly bigger than a

scarcely populated string, and the mean of the luminance is thus relatively higher. This means that we take a smaller part of the peak into account as the amount of visible notes on a string increases. Although this did not hurt performance, as we will see when we describe the derivative analysis we applied to find peaks, we would prefer to have the threshold dependent only on the background brightness. We decided to keep this method of threshold calculation because of the limitations imposed by varying brightness between strings, as discussed at the start of this section.

Derivative alanysis is used to find the local maxima. Peaks caused by notes are characterized by a large positive derivative leading to maxima, as well as a well-correlated width to the template note that was calculated in the convolution step. Note-tails, when present, cause a characteristic pattern after the peak of the note. We use this description of note peaks to build a custom peak detector suited for the specific input.

We find a place in the data where our derivative is bigger than +2 and the datapoint is bigger than the threshold. This is a good indication of the start of a note. We search on through the data until we find a place within half the width of a note into the data where the derivative becomes negative. This allows us to threshold a significant part of the note without the algorithm failing. The zero-derivative point is taken as the location of a peak. We now step approximately half a note width forward and search for the next note. If we do not find a peak within a note width we consider it a false positive and search further for the start of the next note.

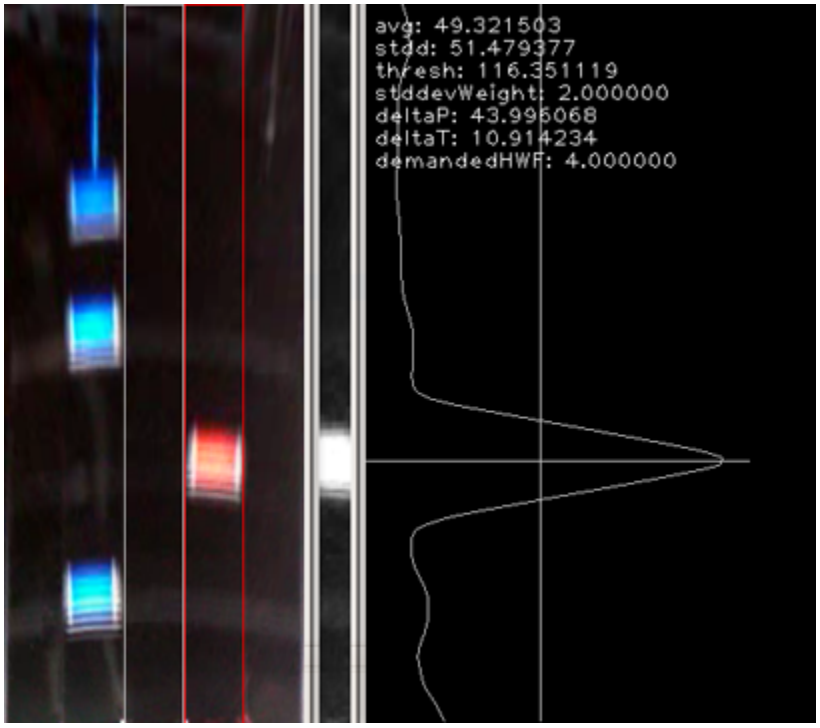


Figure: Example of video input on the left, analysis done on red note, producing luminance plot with threshold and detected peak shown on the right.

Peak Timing

Peak timing estimates how fast notes move across the screen. Peaking timing is performed by correlating detected peak data over successive frames. If a peak is present in one frame, when the next frame is processed the peak is correlated with the first peak past the original position. For robustness, this correlation is done for peaks on each string. All timing data is then averaged over time using an exponential moving average. This is important for simulating the movement of notes in the peak tracker described in the next section; an estimation of note velocity is required for moving data through the peaking tracking circular buffer.

To explain more in detail peak timing is done by finding the rate of change in terms of pixels and time. To find the rate of change in terms of time, we calculate the time when we receive the current frame and the next frame. We get an accurate time by taking the number of computer ticks and converting it into milliseconds. We take these two times and subtract them to find the rate the frame is changing. This is then added to the timing data which as explained before is averaged over time using an exponential moving average. To find the rate of change in terms of pixel, we compare each peak between two frames. We calculate the rate the pixel is moving by subtracting the pixel position between the two frames. After we acquire the rate for each pixel we calculate the average and then once again it is averaged over time using an exponential moving average.

Peak Tracking & Noise Reduction

Once we have detected the peaks, we store the notes in a circular buffer that is designed to simulate their movement. Luminance values for sequential frames are added to the previous buffer values and the result is stored back in the buffer. This accumulation of luminance data creates distinguished peaks and helps to filter out noise. Further peak detection, thresholding, and normalization are then done to extract the note that the cursor should hit. We place a variable-position cursor close to the bottom of our buffer, signifying the point on screen where a hit needs to be generated if a note is present. As the cursor sweeps through buffer, each note it passes sets the guitar state appropriately to cause a note hit in the game. Notes are invalidated as the cursor moves over them. To adjust for camera lag, the cursor can be moved up or down by the user.

Hardware Interface

We can now calculate the state of the guitar at any point in time using our computer vision system. This state now needs to be interfaced to the XBox controller itself. We dismantled an XBox controller to gain access to the control circuitry, where we injected our own interface board. We used the Arduino microcontroller to expose a USB device that our C code can interface with. The Arduino microcontroller stores the guitar state and uses a circuitboard designed and built by our team to toggle the guitar's 5 input buttons and strum control.

The guitar's state consists of the boolean state of its 5 note buttons and its strum toggle switch. Rockband segments the music in songs into a progression of 5 notes, thus the 5 buttons on the guitar. Each note needs to be strummed by hitting the strum toggle switch, adding another input. Thus, our guitar's state as exposed by the Arduino microcontroller is a set of 6 booleans:

state = {red,green,blue,yellow,orange,strum}

Arduino (www.arduino.cc) is an inexpensive prototyping board containing a 16Mhz MIPS processor, a USB device and 13 programmable input/output pins. We use a simple serial communication protocol that packs the state into a single byte and send this to arduino. Arduino unpacks the byte into its 6 boolean values, and switches 6 of its output pins according to the state of the guitar. These 6 pins are taken as input to our interface board.

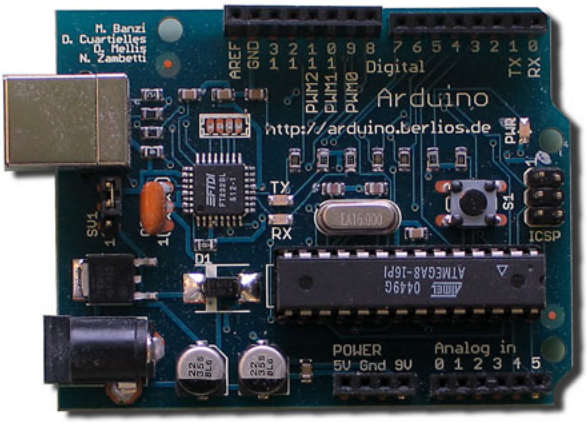


Figure: The Arduino interface board

Our interface board consist of a transistor network connected to the XBox controller's inputs. The transistors are connected to Arduino, causing the output from the aforementioned 6 pins to either ground of float the controller's inputs. Thus, by pulling the transistor's input to high by outputting a 1 from Arduino, the xbox controller's button input gets connected to the controller's "ON" voltage level, simulating a button press. Since this whole process takes very little current, our module is completely powered through USB.

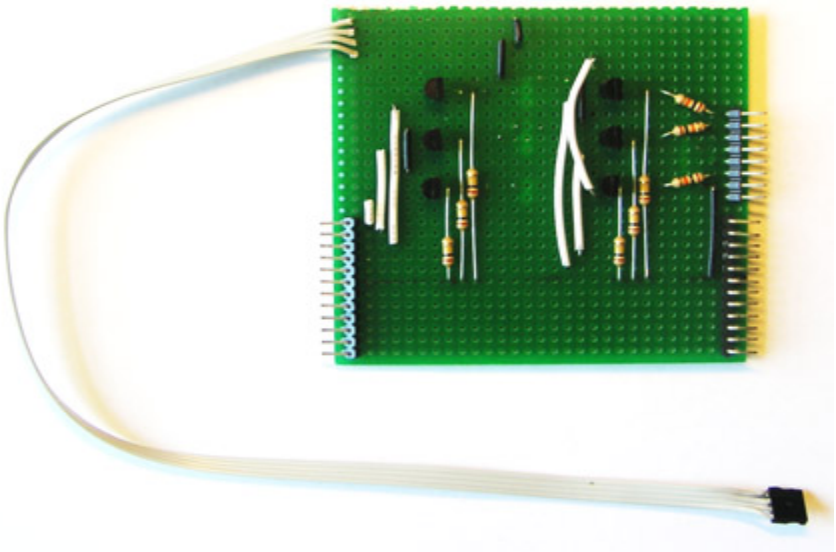


Figure: Our RockBand interface board

We modified the guitar itself by installing 5 light emitting diodes into the neck, corresponding to the 5 note buttons. The output of Arduino is visualized through switching these LEDs on when the

microcontroller pulls an output pin to HIGH, thus displaying the state of the button on the neck of the guitar.

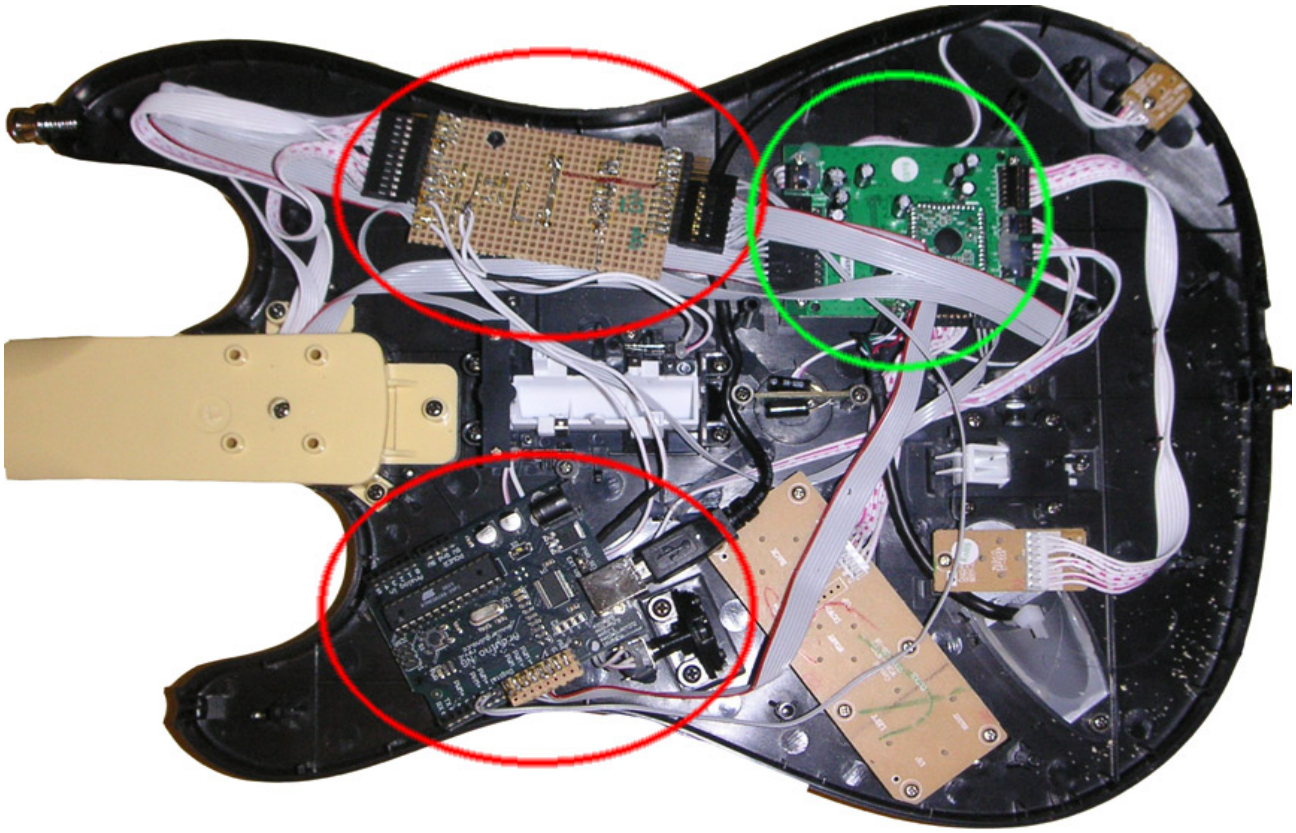


Figure: The completed guitar, with the Arduino microcontroller (red, bottom) and the interface board (red, top) connected to the XBox control circuitry (green).

Figure: Completed XBox Guitar. The only visible change is the 5 LEDs in the neck, providing a visualization of the buttons pressed by our vision system.

Results

The final product meets our design criteria and solves the challenge. We can play the Rockband game successfully, reaching up to 96% accuracy on Expert-level songs. Currently out only limitation is that we cannot play notes with tails. Our team collaborated well, with each person being responsible for a crucial part of the design and implementation. We are pleased with our result, and identified several areas where we would want to expand on this project in the future. We ask the user to configure two things: the perspective transform that selects the fretboard, and the cursor that accounts for camera lag. The perspective transform can be done automatically by edge detecting the note board instead of through a GUI. Cursor adjustment can be automated through a feedback control loop if we can find a data feed giving us information about how close a note hit was. This is possible on the Nintendo Wii through a control signal, but on the XBox we would have to revert to detecting note hits visually as they create effects on screen.

We enjoyed this project tremendously. We will be releasing the source code and accompanying materials online in the near future.

Accompanying Media

Included with this report is a CD-Rom containing video footage of the project. We show how it was built and how it performs. This report and video will also be made available online at http://inst.eecs.berkeley.edu/~njoubert/

References

Slashbot `http://slashbot.wordpress.com/`
AutoGuitarHero `http://www.autoguitarhero.com/`
OpenCV `http://opencvlibrary.sourceforge.net/`

Contact Details

Niels Joubert, CS184-DV, niels@berkeley.edu, njoubert@gmail.com
Rohit Nambiar, CS184-AJ, jedirohit007@berkeley.edu
Navin Lal, CS184-AK, njlal@berkeley.edu
Mark Sandstrom, CS184-DH, smark@berkeley.edu, mark@deliciouslynerdy.com