ImprovTetris Proposal

Scott Bezek
Ray Li

1 Overview

In the classic version of Tetris, the player must orient and piece together randomly-picked falling blocks in an attempt to complete and clear entire rows before everything piles up to the top. ImprovTetris aims to implement the game of Tetris on an FPGA, but with an improvisational twist: instead of just using the standard seven Tetris blocks, ImprovTetris allows players to make up their own custom blocks on the fly. Furthermore, these custom block shapes are defined by the player's actual physical stance; the player uses his body to control the game, much like a Microsoft Kinect.

In order to provide this real-world control interface, ImprovTetris uses a camera to capture video of the player, which is then analyzed by the FPGA to generate a quantized Tetris piece based on the silhouette. An overview of the analysis process is shown in Figure 1, and is detailed in depth in Section 2.1.

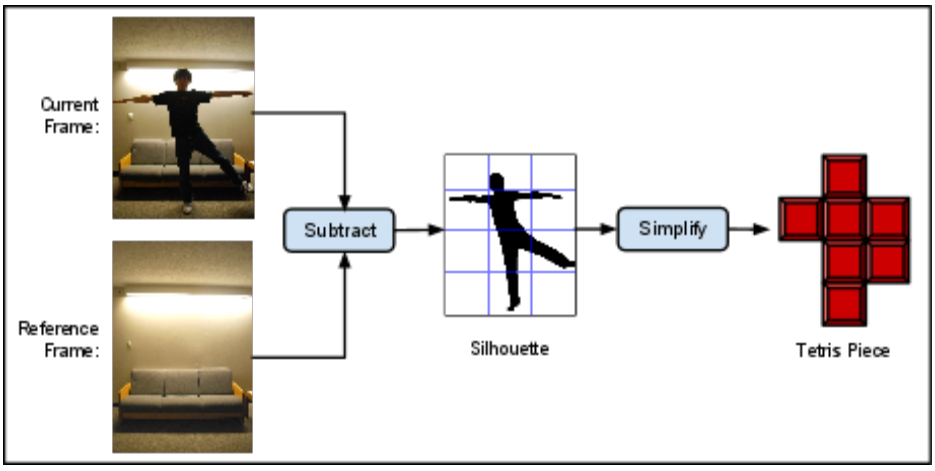

Figure 1: ImprovTetris analyzes live video and simplifies the player's body shape to generate custom Tetris pieces.

While the block is falling, the player can contort his body to change the block's shape in real-time. This allows him to create blocks that perfectly fit into the existing playing field, or he can even squeeze a block through a small crevice before expanding it to fill the open space. ImprovTetris also provides on-screen feedback of the player's silhouette and the simplified block shape to help him form useful pieces - a display mock-up is presented in Figure 2.

The rest of the game-play is standard Tetris: the current block falls at a constant rate, and the user can slide it left or right by pressing buttons on the FPGA. When an entire row is filled, it disappears and the remaining blocks shift downward. The player scores by clearing rows, and can earn higher scores by clearing multiple rows at once. If the playing field fills up and there is no remaining space for a block to fall, the game ends.

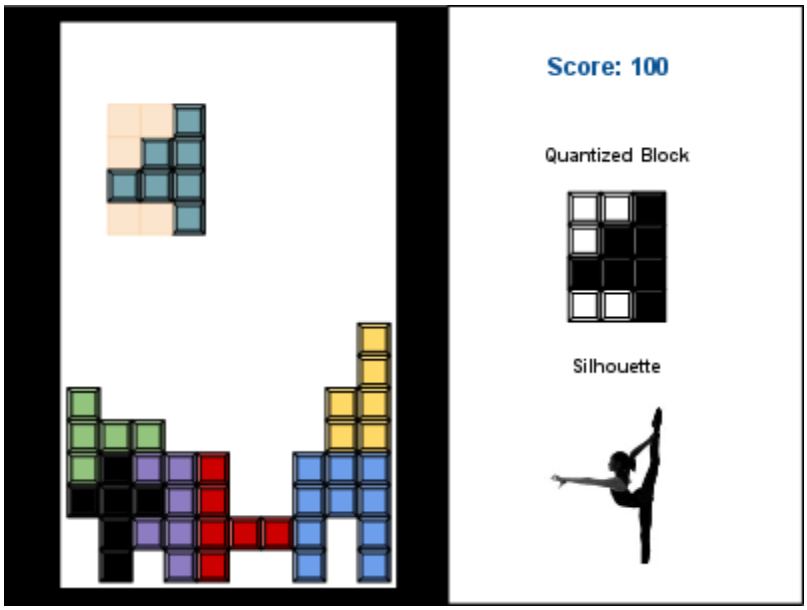

Figure 2: Mock-up of the ImprovTetris display: Tetris playing board on the left, with the score board, silhouette, and quantized block shape on the right.

2 Implementation

The implementation of ImprovTetris is broken down into three major subsystems: Image-Processing, the finite state machine logic (FSM), and the Display Module (Figure 3). The Image-Processing subsystem takes in image frames from the camera and outputs a 3x4 grid of the player's block shape to the FSM. The FSM controls the game logic and updates the playing field: it causes the current block to fall, detects when the block has settled, and clears/scores any complete rows. The Display Module takes the silhouette, the quantized Tetris piece, the playing field and score, and images from the ROM to piece together the display output. The following sections describe how these subsystems function and interact.

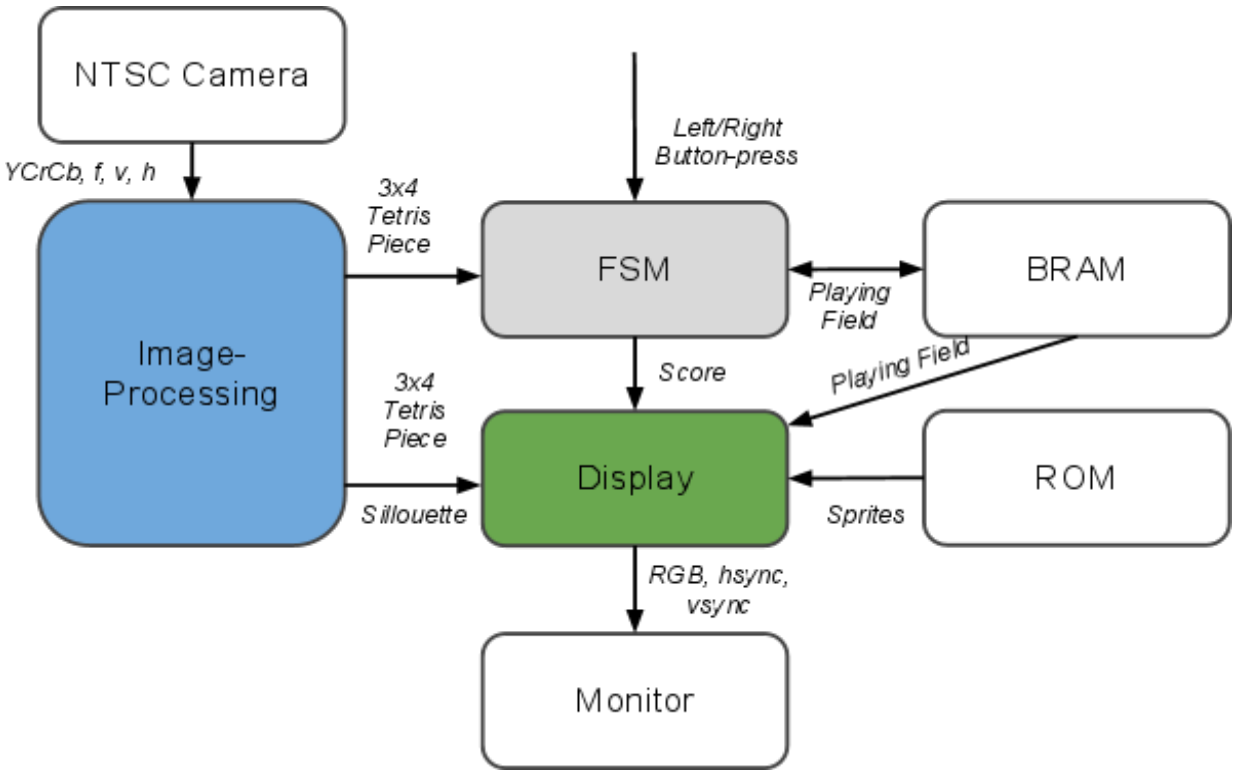

Figure 3: Block Diagram of ImprovTetris, highlighting the three major subsystems.

2.1 Image-processing Subsystem (Scott)

The image-processing subsystem is responsible for capturing NTSC video, filtering, down-sampling, and quantization to determine what block shape the player's body forms.

The basic process is to capture and save a reference image at the beginning of the game when the player is not standing in front of the camera, and then compare each video frame during the game to this reference image (as shown previously in Figure 1). If any pixel differs enough between the reference frame and the current frame, then that pixel is considered "occupied." These occupied pixels form a silhouette of the player, which is then quantized into a 3x4 grid - this is the custom Tetris piece that falls in the game.

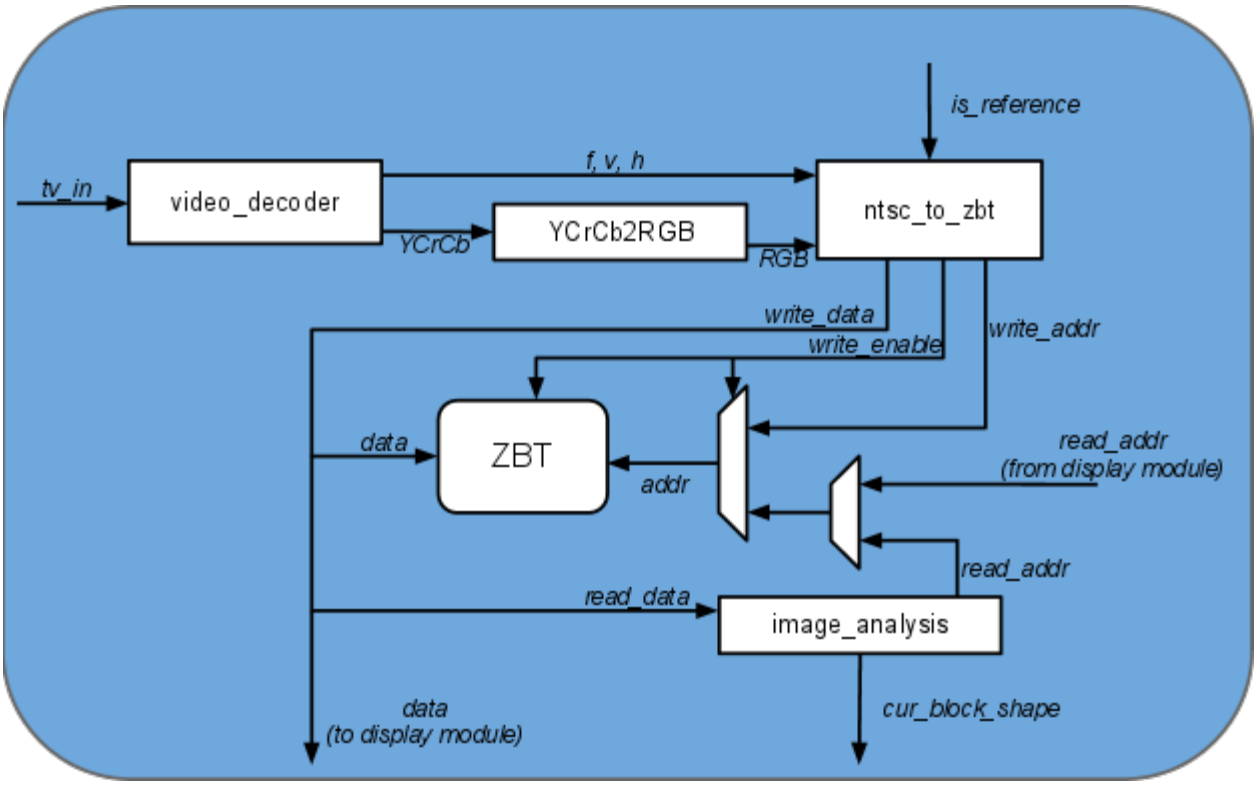

Figure 4: Block diagram of modules within the Image-Processing subsystem

The Image-Processing subsystem (Figure 4) is composed of the following modules:

1. **video_decoder** - [Provided by 6.111 staff] Accepts signals from an NTSC camera and decodes that information into YCrCb, vertical and horizontal sync signals, and a field bit which denotes even vs. odd interlaced fields
2. **YCrCb2RGB** - [Provided by 6.111 staff] Converts the 30 bit YCrCb data into an 18 bit RGB value.
3. **ntsc_to_zbt** - Stores each frame of NTSC video (as 18-bit RGB values) to ZBT memory
4. **image_analysis** - Reads the current video frame out of memory and compares it to the reference frame. Calculates the silhouette and outputs a 3x4 Tetris piece

2.1.1 The ntsc_to_zbt module

One of the major challenges of the image-processing subsystem is storing and accessing the camera images in memory. ZBT memory cannot simultaneously read and write at different addresses, so we need to carefully specify when image data is being stored vs. accessed. There are also constraints on the amount of data that the ZBT memory can hold.

This module accepts the horizontal/vertical/field signals from the video_decoder along with the 18-bit RGB values for the current pixel and outputs the memory address and data to write to memory. Since each pixel takes up 18 bits, we will store 2 pixels per memory address. With 512x512 pixels of video data, this will use 131,072 addresses in the ZBT memory chip, which has a total of 512K addresses (~25% usage). However, since we will be storing a reference frame along with the current frame, this will use a total of 262,144 addresses.

Inputs:

1. f, v, h - The field, vertical, and horizontal sync for NTSC
2. RGB - The 18 bit RGB value for the current pixel
3. is_reference - Determines whether or not to save this as the reference image

Outputs:

1. write_addr - The address to write in ZBT
2. write_data - The data to write in ZBT (2 pixels of RGB data)
3. write_enable - Controls write access to the ZBT

Testing Strategy:
This module can be tested in simulation by supplying some fake NTSC data and making sure that it outputs the correct memory addresses and combined data for each pixel that is input.

2.1.2 The image_analysis module

This module reads the current frame and reference frame out of ZBT memory, does some simple filtering, and calculates the "silhouette," It uses the silhouette to calculate the custom 3x4 Tetris piece, which it outputs to the main FSM and display modules.

This image processing is done sequentially (i.e. pixel-by-pixel) during one frame of the VGA display. It will take on the order of 262,144 cycles to process the images (there are 512*512 pixels). Rendering the VGA frame takes ~600,000 cycles which should provide plenty of time for the processing to take place in parallel.

Inputs:

1. read_data - The data being read from ZBT

Outputs:

1. read_addr - The address to read a pixel from the ZBT memory
2. cur_block_shape - The 12-bit representation of the calculated 3x4 Tetris piece

Testing Strategy:
This module will need to be tested on the FPGA due to the complexity of the signals involved. To debug and verify correct operation, it is possible to use the BRAM on the labkit in order to store the calculated silhouette and output this to the VGA display.

2.2 FSM (Ray)

The FSM handles the general game logic and keeps track of the scores. The Tetris game starts off in the *Falling* state when the player's custom block is falling down. When it lands, the game goes into the *Clearing* state to identify the complete rows, then goes through a flashing animation in the *Flash* state, and finally shift the blocks down in the *Collapse* state to remove the cleared rows. If nothing can be cleared and the most recent piece hits the top of the playing field, then the game end in the *Game_Over* state. The player may reset and initiate a new game, which starts over in the *Falling* state. The state diagram is shown in Figure 5.

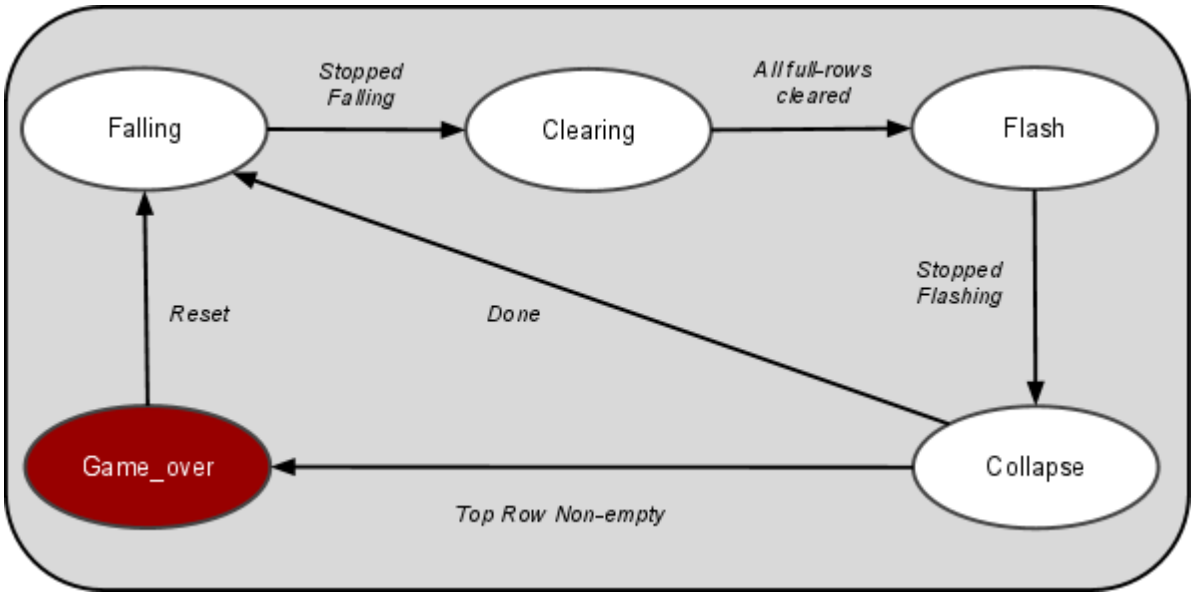

Figure 5 - Diagram of FSM state machine that governs the game logic.

States:

1. Falling - piece is falling down
2. Clearing - after falling piece lands, this marks full rows to be cleared
3. Flash - flashing animation for rows marked to be cleared
4. Collapse - collapse rows marked as cleared and calculate score
5. Game_Over - after grid piles up to the top, freeze animation and show score.

This module relies on BRAM to store the state of the playing field. The playing field is a 10x20 grid, where each grid cell may be occupied by one square of a colored Tetris piece. Three bits are used to represent the color of each cell, with one of eight possible values reserved to represent empty cells. Each row of the playing field is stored within a 32-bit word in BRAM (the two extra bits can be used to keep track of animation and clearing of the row). There are two playing field grids stored in BRAM: one contains the "settled" pieces, while the other contains only the falling piece. These two grids are combined by the Display Module when drawing the output.

Inputs:

1. Shape (3x4 grid) of player's quantized silhouette
2. Left/right button presses
3. Game State (BRAM) - location of blocks

Outputs:

1. Score (to display)
2. Game State (BRAM) - updated location of blocks

Testing strategy:
To test this module, we will specify the shape of the falling block through the switches[7:0] and the left/right movement with button_left and button_right. This should be enough to test all the states of the game to check for correctness.

2.3 Display Module (Ray)

The display pieces together the game state from the BRAM, sprites from the ROM, the silhouette from the SRAM, the score from the FSM, and the calculated quantized block shape. These elements are combined on a pixel-by-pixel basis so that the display module can output the appropriate signals to the computer monitor.

Inputs:

1. Game State (BRAM) - render the blocks in grid
2. ROM - render fancy images for styling
3. Score (from FSM) - render score
4. SRAM - render silhouette of player
5. Image-Processing - render the quantized shape of the falling block

Outputs:

1. Monitor

Testing strategy:
We can just read out solid blocks of different colors to represent each input component and output it to the monitor.

3 External Hardware

ImprovTetris requires minimal external hardware - only an NTSC video camera is needed to capture the movements of the player. All of the image processing and game logic can be implemented directly on the FPGA labkit.

4 Possible Extensions

There are a number of possible extensions to ImprovTetris that can adjust the difficulty or incorporate other forms of user input. For example, instead of using buttons to control the falling block's left/right motion, we could attach a gyroscope sensor to the player and require him to lean left or right to move the block (this could theoretically be done directly through image analysis, but would likely require more computational power than is available on the FPGA). Another possibility is to add interesting sound effects to the game when the player clears a row or for very high-scores. A more ambitious extension possibility would be to network two FPGAs to support a 2-player ImprovTetris game. Depending on the amount of time available, it may be possible to implement some of these ideas into the final product.