Adversarial Tetris Preliminary Report

Aishah Jones
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
aishah@mit.edu

Elmer Cruz
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
cruze@mit.edu

***Abstract*—Tetris is a classic arcade game where a single player must destroy lines of blocks before said blocks reach the top of the screen. The active, player-controlled block is randomly chosen from a set of blocks, and the player can move the block side-to-side and rotate the block at will while the block is "falling" down the screen. Once the block lands at the bottom of the screen or onto another block, it becomes immovable, and a new block is generated for the player to control. Our project implements an extension of this game where the next block is generated by a second, adversarial player by means of holding an image of the desired block up to a camera. The adversarial player has the ability to generate either an existing Tetris block or create a "custom" block that can add varying levels of difficulty for the active player.**

I. Introduction

In order to implement Adversarial Tetris, the system is divided into three major groups: Game Logic, Graphics, and Camera Image Processing. This report will detail the specifications of each major group, and a block diagram of the entire system can be found on the final page of this report.

II. Game Logic

The Game Logic portion of the system is responsible for initializing and updating the Tetris game states, and its outputs will be sent to the Graphics portion in order to display the game onto a computer monitor. An outline of the Game Logic subsystem's states, inputs, and outputs are given below:

States:

- Welcome Menu
- Play Game
- Pause Game

Inputs:

- FPGA Button Signals
  - btnl
  - btnr
  - btnc
  - btnu
  - btnd
- `active_board[3:0][3:0][11:0]`
- `active_board_dimension[1:0][2:0]`

Outputs:

- `new_game`
- `landed`
- `game_state([2:0])`
- `out_board[19:0][9:0][11:0]`

The Game Logic portion of the system will be responsible for initializing and updating the Tetris game states, and its outputs will be sent to the Graphics portion in order to display the game onto a computer monitor. The Game Logic module starts with a Welcome Menu state where the signals received from the up and down buttons help navigate the menu and the center button makes selections between regular and adversarial Tetris. Once selected, the state changes to a Play Game state where a block starts to descend, and the player can control the movement with the left and right buttons. The up button in this state will serve with rotating the block, the center button will be used for switching between a paused and a playing state and the down button will serve as the restart button which returns the game to the initial state. In the playing state, besides the button signals as inputs, the module also uses an *active_block* input along with *active_block_dimensions* input. *active_block* is a 4x4x12 array that gets passed in from the *next_block_generator* module and this array represents the playable block in the game. *active_block_dimensions* is a 2x3 array that gives the dimensions of the block since not every block fills the 4x4 space. The output of this module is the updated board after every move made by the player. The logic uses the board to determine if a move can be made when that movement's respective button is pressed and it updated the board with this moved active block. The *landed* output of this module tells

the next block generating module to start generating the next block. The *new_game* output specifically differentiates between the game intro at the start of a game and the intro during pauses. The *game_state* output keeps track of the game FSM for the rest of the build.

III. Graphics

The Graphics subsystem is responsible for displaying all necessary visuals onto the computer monitor. The computer monitor itself displays three things: the Tetris game board, a preview of the next tetromino that will fall, and a live camera feed. Figure 1 shows a mockup sketch of this layout.

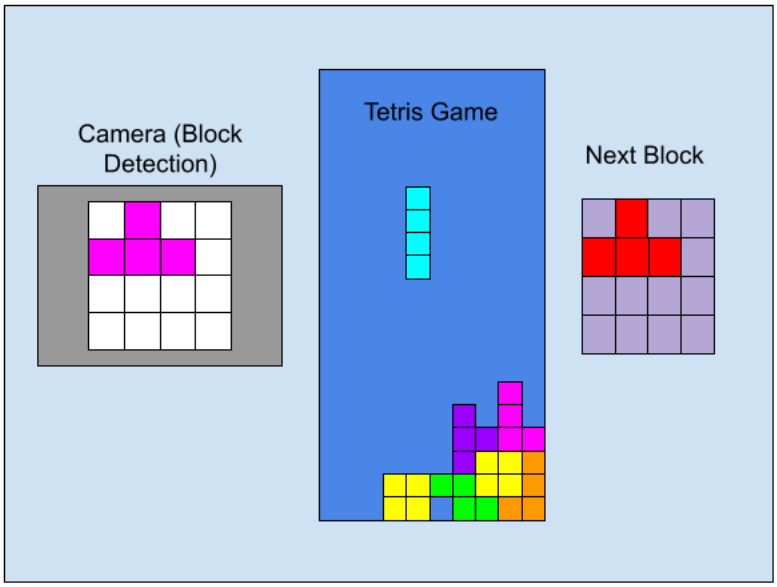

Fig. 1. A mockup of what is to be displayed on the computer monitor during gameplay

The Graphics subsystem will thus be split into two modules for the game board and next tetromino preview, and a smaller subsystem to encompass the camera feed. A multiplexor at the end of the Graphics system will, based on the VGA's *hcount* and *vcount* values, determine which pixel from each subsystem (if any) is displayed on the monitor.

The following inputs will be routed to each subsystem in Graphics (and thus not explicitly repeated in input listings on future pages):

- `65MHz clock`
- `hcount[10:0]`
- `vcount[9:0]`

A. display_board module

The display_board module is responsible for converting the grid coordinates from Game Logic into pixel coordinates and displaying those pixels onto the monitor. The game board is a 10x20 grid where each grid square is 30 pixels long. In pixels, the game board is 300x600.

Inputs:

- Graphics system inputs
- `board_in[19:0][9:0][11:0]`: A 10x20 array of 12-bit numbers representing the current game board (output of Game Logic)
- `current_game_state`: A number representing one of the 6 game states

Outputs:

- `pixel_out[11:0]`: A 12-bit number representing the [R,G,B] value of the game board pixel at (*hcount, vcount*)

B. next_tetro_preview module

The next_tetro_preview module is responsible for showing a preview of the next tetromino to fall on the monitor. This is a standard feature of the classic Tetris game that allows the player to quickly plan where they will place that block. For our system specifically, this feature also serves as feedback to confirm that the Camera Image Processing subsystem correctly identified the block being displayed in front of the camera.

Inputs:

- Graphics system inputs
- `next_tetro[3:0][3:0]`: A 4x4 grid representing a tetromino. A value of 1 in a given array location indicates that that grid square is part of the tetromino. This input comes from the Camera Image Processing subsystem.
- `current_game_state`: A number representing one of the 6 game states

Outputs:

- `tetro_pixel_out[11:0]`: A 12-bit number representing the [R,G,B] value of the block preview pixel at (*hcount, vcount*)

C. Camera

The Camera subsystem is responsible for accepting input directly from a camera connected to the FPGA and displaying the masked image on the screen. The system applies a red chrominance mask

to the camera image, thus the adversarial player must show bright pink/red colored tetrominoes to the camera for proper detection. The objective is that the visual of the mask on screen will provide clear feedback on how easily and clearly the blocks are being isolated by the system. This subsystem is essentially a simplified version of the infrastructure provided for Lab 4, and a block diagram is provided at the end of this report.

Important modifications and notes regarding the given infrastructure are as follows:

- `scale`: This module has been removed due to the fact that the camera image on screen will be at the smallest scale, 240x320.
- `mirror`: Takes no switch inputs; mirror will always be on for our system
- `rgb_to_ycrb`: Modified so that there is only one output: cr_in
- `threshold`: Modified so that there is only one input: cr_in
- `vga_mux`: Removed and replaced with combinational logic in top_level in the project's current iteration; will likely be returned at a later stage to increase top_level's readability

IV. Camera Image Processing

The Camera Image Processing subsystem is responsible for analyzing the camera data output from the Camera subsystem and determining what tetromino is currently being shown to the camera. The accuracy of this subsystem is essential for smooth, enjoyable, and functional gameplay. As demonstrated during the class labs, the existing red chrominance mask performs exceedingly well at isolating red or pink objects, so this system's design will use that to its advantage.

A. tetro_detector module

The Camera Image Processing subsystem is encapsulated entirely in the *tetro_detector* module. Any given tetromino can be thought of as existing on a 4x4 grid - the squares making up the tetromino are considered "active", and the empty squares are considered "inactive". This module determines the next tetromino by keeping 16 running sums, one for each square of the 4x4 grid, and updating these sums based on *hcount*, *vcount*, and the masked camera feed input coming from the Camera subsystem. If this pixel input equals the system's mask value (currently this value is hexadecimal number A26), then the sum for the square containing that pixel is incremented by 1. If the sum for a square equals the area of that square (i.e. length x width in pixel coordinates) to within a small margin, then it is considered an active square.

Inputs:

- `65MHz clock`
- `hcount_in`
- `vcount_in`
- `masked_pixel_in[11:0]`: A 12-bit number representing the [R,G,B] value of the masked camera pixel at (*hcount, vcount*)

Outputs:

- `next_tetro[3:0][3:0]`: A 4x4 array representing a tetromino block. (A 1 represents an "active" square, a 0 represents an "inactive" square)

V. Next Block Generator

The Next Block Generator subsystem will be in charge of converting the array received from Camera Image Processing and converting it into a Tetris block by assigning each detected sub-block the same randomly assigned color. If adversarial game mode is not active or if no block is detected, then the next block that gets generated is randomly selected from the seven default Tetrominos as shown in Figure 1. The color is randomly selected regardless for this version of Tetris.

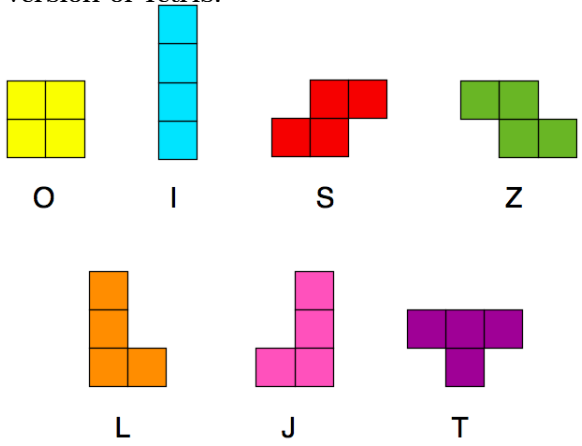

Fig. 2. The seven Tetrominoes, each named according to the alphabet letter they most resemble.

Inputs:

- Camera Block: A 4x4 array received from Camera Image Processing

Outputs:

- `next_block[3:0][3:0][11:0]`: Represents the next playable block
- `next_block_dimensions[1:0][2:0]`: Representing the dimensions of the Next Block since the block sizes aren't constant or always fill up the 4x4 space

VI. Evaluation

This system ran into many trade-offs, especially between throughput and resource usage. Because

Tetris relies on being able to determine whether a move can be made or not relative to other blocks and the borders of the board, it became easier to use large structures like our 20x10x12 array representing the board. Arrays have the benefit of being readily accessible which simplified the logic for block movements and rotations.

VII. Block Diagrams

See **System Overview** and **Camera Subsystem** block diagrams on the following pages:

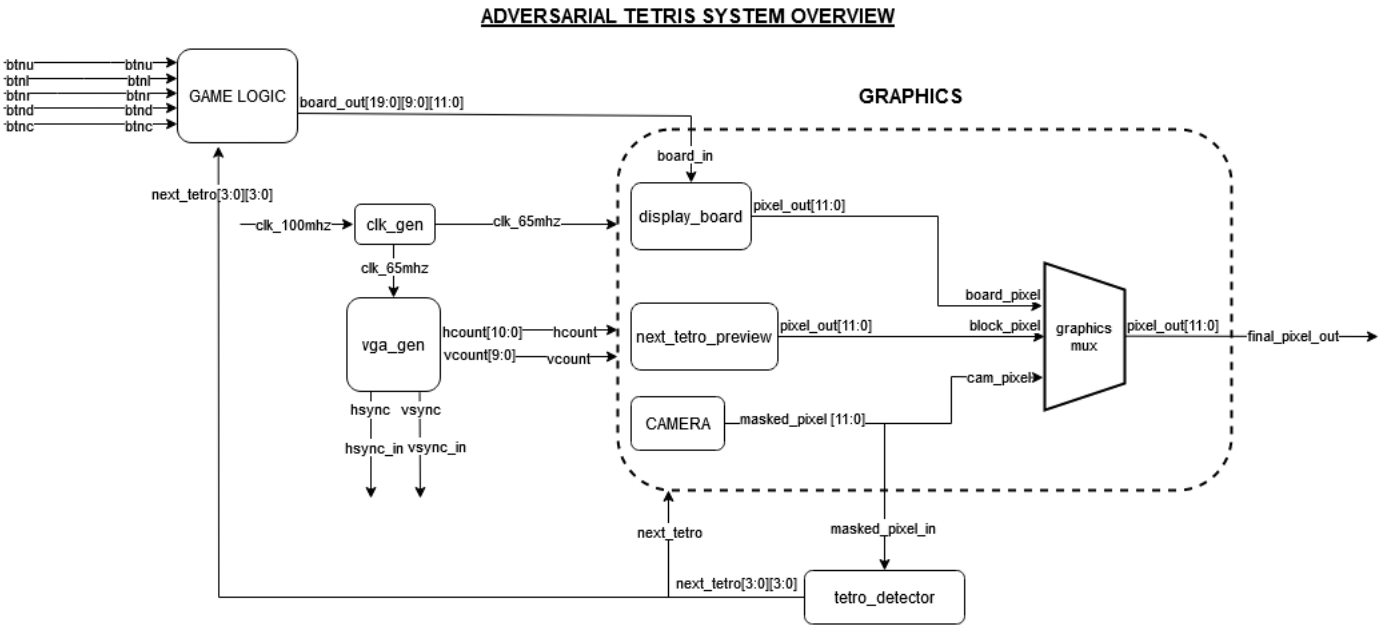

Fig. 3. Block diagram representing the entire Adversarial Tetris system

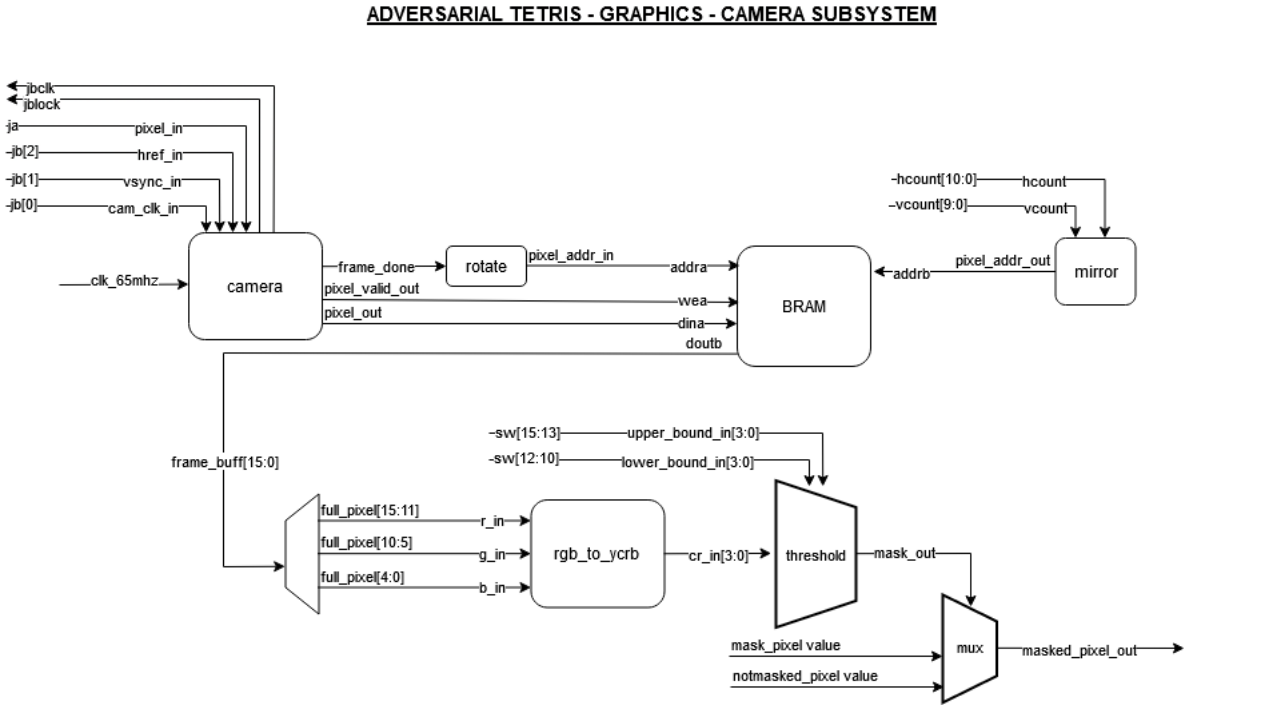

Fig. 4. Block diagram for the Camera subsystem responsible for displaying live camera feed with a red chrominance mask