# CSE 165 Final Project Report

Hunter McClellan Higinio Ramirez Kevin Zheng

May 9, 2021

## 1 Project Description

Our project is recreating the famous Tetris game in 3D. Rather than clearing "lines", the player is expected to clear layers. The pieces will be reminiscent of the original pieces, since most of them can be quite naturally generalized.

## 2 Member Contribution

The contribution can be roughly be separated by file as as

- Hunter McClellan
  1. `mainwindow.h`
  2. `mainwindow.cpp`
  3. `Ground.h`
- Higinio Ramirez
  1. `Points.cpp`
  2. `Randomizer.cpp`
- Kevin Zheng
  1. `Tetris.h`
  2. `Tetris_Graphics.h`

## 3 Implementation

### 3.1 Tetris.h

Handles all game logic

- `GameState` enum: has only two members, `PLAYING` and `LOSE`. Sorry, you can't win this game!
- `Moves` enum: has eight members, `DOWN`, `LEFT`, `RIGHT`, `FORWARD`, `BACK`, `PITCH`, `ROLL`, `YAW`.
- `Block` struct: has three integers for red, green, blue which range from 0 to 255. The `Block` struct also has one bool that indicates whether or not the block is falling. Falling in this context is synonymous with "in play." Note that the location of the `Block` is not handled by the `Block`, it is handled by the `Tetris` class.
- `Tetris` class: has a quadruple `Block` pointer denoted by `state`. This can be best understood as three dimensional array of `Block` pointers. This number of pointers is necessary, since want to allow parts of the array to be `NULL`. There are three integers for width, length, and height of the playing field.
  - `ind2sub(int ind, int &x, int &y, int &z)`. Short for "index to subscript". This method accepts a number from 0 to `w*l*h-1`, and "returns by reference" the corresponding `x`, `y`, `z` coordinates. This is used in the cases where we want to loop through every `Block` in `state`, but would rather not use a triple for loop. Instead, we can make one loop, and use `ind2sub`.
  - `int sub2ind(int x, int y, int z)`: This is the reverse operation of `ind2sub`.
  - `GameState control(Moves moves)` accepts a member of the `Move` enum and modifies the `state` accordingly. If the control made by the player results in a loss or not will be reflected in the `GameState` return. The body of this method is simply a `switch` statement calling either `translate_piece` or `rotate_piece`.
  - `GameState translate_piece(Moves move)` handles the translation of pieces. This is called by `advance` as well as `control`
  - `void rotate_piece(Moves move)` rotates the piece. Note that the return is `void` since a rotation can never cause a loss.
  - `void handle_layer_clear()` Checks for cleared layers, and remove them.
  - `GameState spawn_piece()` attempts to spawn in a random piece at the top of the playing field. Failure to due so is the definition of a loss, and such will be returned.
  - `GameState advance()` is an alias for `translate_piece(DOWN)`, as well as a `handle_layer_clear()`.

### 3.2 Tetris_Graphics.h

Has a constructor which accepts a `Tetris*`. Has a `void draw()` method which draws the Blocks in `Tetris`.

### 3.3 mainwindow.h/mainwindow.cpp

Handles keyboard input, mouse input, and graphics. Has class `MainWindow` which extends `QOpenGLWindow`. Implementes the generic methods of `QOpenGLWindow` like in the labs. Adds a `SLOT` for the method `GameAdvance()` which is an alias for `Tetris.advance()` which is bound to a `QTimer` for once a second.

- `void paintGL()`. Starts by clearing the `GL_COLOR_BUFFER_BIT` and the `GL_DEPTH_BUFFER_BIT` and calls `glLoadIdentity`. Then we do a `gl_Translate(0.0, 0.0, -35.0)`, which elevates the camera by `35.0`, this is so the player has a somewhat downward facing view of the playing field. Then we call `glRotatef` for each direction. Then, we can call `Tetris_Graphics.draw()` and `Ground.draw()`, to draw the ground and the game pieces. Afterwards, the controls are handled by checking for if the `W,A,S,D`, $\uparrow$, $\downarrow$, $\leftarrow$, $\rightarrow$ keys have been pressed, and then make the corresponding call to `Tetris.control()`.
- `void keyPressEvent(QKeyEvent *event)` checks which of the `W,A,S,D`, $\uparrow$, $\downarrow$, $\leftarrow$, $\rightarrow$ keys have been pressed, and then sets the corresponding member boolean to be true. The escape key calls `qApp->exit()`to exit the game. Note that these member booleans are also set to false after `paintGL` as to prevent the game from doing the move twice before release.
- `void keyReleaseEvent(QKeyEvent *event)` does the opposite of `keyPressEvent`.
- `void mouseMoveEvent(QMouseEvent *event)` modifies the `cam_x_r, cam_y_r, cam_z_r` according to the position of the mouse. This method also resets the position of the mouse to the center. This is to prevent the mouse to leave the window as the player moves the mouse for a different perspective. To regain normal use of the cursor, exit the game.

### 3.4 Points.cpp

The points system is based on the original NES Tetris scoring system

- Points calculated: has to calculate the points given through layers cleared and by the level that we are currently on.
  - The max layers cleared are four and this is just like normal Tetris where you will get the most points for completing a 4x4x8 layer. First, we need to get the level, layers that are being cleared so we know which points system to use. From this point, we have abased level 0 with a certain amount of points for each section. Every increase in layers cleared will give more than double the previous layer as a way to incentivize staking as many layers as you can before you clear. We start with 40 points at level 0 and double with every signal level increase. The points themselves are 40,100,300, and finally 1200. The equation that we used is 40(level + 1) and just change it depending on the layers being cleared.
  - In order to keep the points for later as we would be continuously adding on points as the game progressed, we just used a reference to points and check the points and add whatever we need to from the game.
- Soft drop: Pressing down the key to dropping a Tetris piece a little faster and again a point for every section drop.
  - In the classic Tetris model, you would get points for every piece that you drop faster than the set speed. These points do not scale up with the Tetris levels as it shouldn't be a real game-changer when it comes down to clearing layers. If two people are playing together at the same time with the same pieces, this extra feature will incentivize a more aggressive playstyle that might make some people overestimate their abilities and drop out faster. The opposite is also true because if a person is good enough they can get a significant amount of points for going faster. Also in the lower levels, the pieces that drop can be relatively slow to some people, so to counteract that soft dropping allows the play to play at a relatively faster rate and enjoy a faster and therefore better game as the player will not have to wait for a pieces to drop to get another piece on the board.

### 3.5 Randomizer.cpp

A basic randomizer for the Tetris pieces that come in to play

- How the randomizer works: We take the number of pieces and us srand(time(0)); to set up the random times.
  - While making this project the randomizers that I made only contained rand() % 7 +1 to get the values of 1 through 7, but I ran into a problem as the values that I was getting were not truly random as the machine was giving random numbers, but keeping that same location of random numbers so we will not be getting truly random numbers. This is where srand(time(0)) comes into play, it allows a truly random set of numbers to be chosen per game.

- Wanting more randomization: Based on an NES Tetris game I have played
  - While making the randomizer I remembered that in NES Tetris there was randomization, but they did something else to have even more randomization. I wanted to make sure that two separate pieces wouldn't have a higher probability to be different. In order to achieve this, I made a check function where the first value of the randomizer would be saved to then check a second randomizer if the value was the same as the previous piece that was given. If this is true then we will activate another randomizer that will give us another number to use and if this number is the same it wouldn't matter at this point as you might have been destained to get that certain piece, but if you get a different number then we will use that new number and save it for the check later on.

## 4 Lessons/Conclusions

### 4.1 Depth Bit

The depth bit, or depth buffer, is a grayscale image. Unlike a regular grayscale image, it does not store brightness, rather it stores the depth of all of the colors on screen. This way, if another object is attempted to be drawn on the screen, it will compare it's depth to the depth buffer, and only write if it's visible.

Not all systems will allocate space for the depth bit, as on a Linux machine, the depth buffer did not work unless the buffer was explicitly allocated. On Windows machines, the depth buffer has a default allocation, so the allocation function was not required.