College of Engineering
COMP 410 – Computer Graphics Project Report

KUTRIS

Participant information:

Begüm Şen 72840
Berat Karayılan 72690
Sinan Cem Erdoğan 68912

Spring 2023

Table of Contents

1. Kutris

Kutris is a Tetris-like game that challenges players to construct specific patterns by strategically placing blocks. The motivation behind developing this game was to create an enjoyable gaming experience while incorporating all the concepts taught in the class. Our goal was to integrate and apply the learned concepts effectively, resulting in a game that is not only entertaining but also serves as a practical showcase of our knowledge and skills.

1.1. Introduction

Each game begins with the random assignment of one of two patterns: 'Home' or 'Heart' for the player to follow (see Fig. 1 and Fig. 2). While it is harder to complete the ‘Heart’ pattern, ‘Home’ requires less intuition.

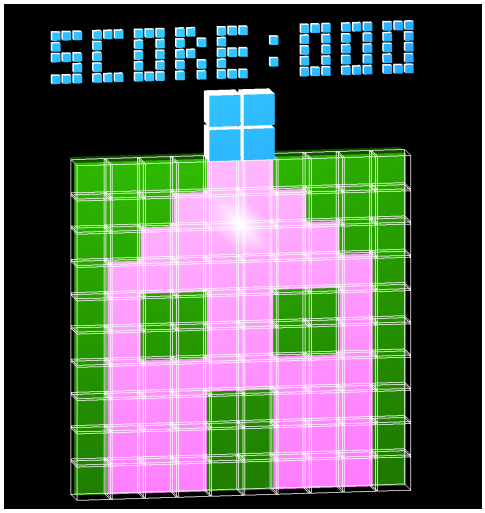

Figure 1 House Pattern

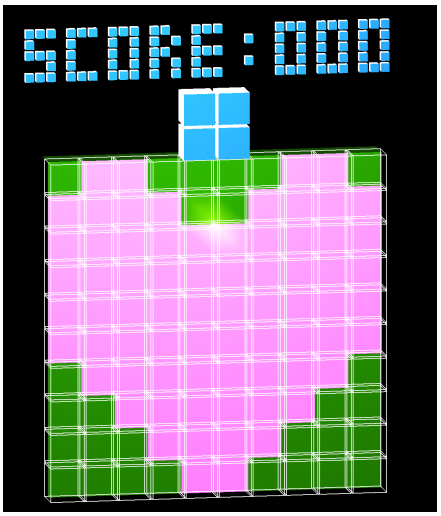

Figure 2 Heart Pattern

To complete the given pattern, players are provided four distinct block types: the 2x2 block, 1x2 block, 1x1 block, and L block (see Fig. 3). Once the starting block has been placed by the player, a new block is randomly drawn on top of the game board between 4 types . The game board is a 10x10 grid.

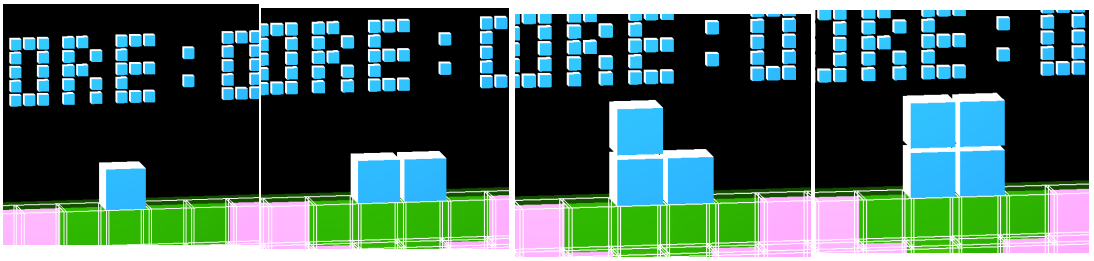

Figure 3 Differently Shaped Block

1.2. Gameplay

There are five actions that can be performed by the player. First one is moving the block one unit at a time. Players can use right and left arrow keys to move the object to the desired place before making it fall. Additionally, the player can rotate the object by pressing the space button, which will rotate the current block by 90 degrees clockwise. (see Fig. 4)

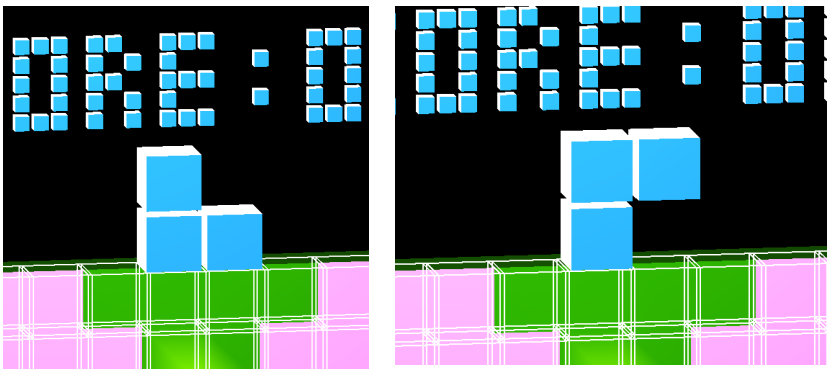

Figure 4 Rotating Blocks

Once the adjustments on the top of the grid are completed, the player places the current block by pressing the arrow down button which will make the object fall from top to bottom. To end the game, the player can press the E key whenever they wish. When the game ends, a "Play Again" button appears for the player to click and start a new game.

1.3. Rules

Each player is given a 5 chance to delete a particular cube from the current cube before placing it. In that way, when the player is stuck, meaning they could not find a place to make the block to fall, they can change the block type by deleting the selected block. Selection is performed by clicking on the cube with the mouse. (see Fig. 5)

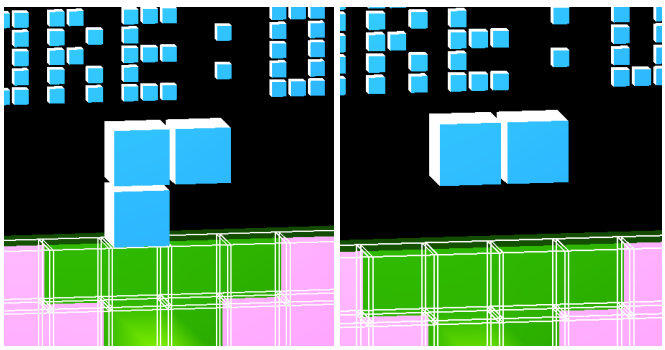

Figure 5 Deleting Cubes from Block

1.4. Scoring

Score is calculated as follows; for each block placed correctly to the pattern the player gets 1 point and loses 2 points for any wrongly placed one. However the score cannot be below 0 thus it is the lowest score that the player can have. (see Fig. 6)

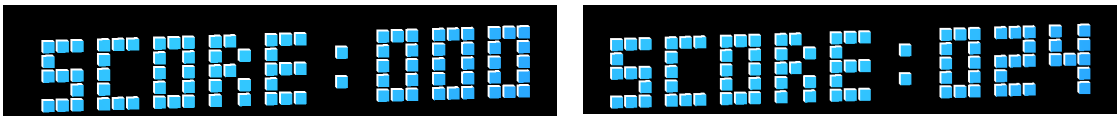

Figure 6 Score

2. Concepts

2.1. Basics

To draw objects to the scene, shader coding and 3D modeling is used. At the beginning of the game, 10 of each block type are placed in the buffer. The number of cube required to make the blocks are calculated as follows (refer to Appendix A):

- 1x1 block requires 1 cube, in total 10. The first 10 vaos belong to the 1x1.
- 1x2 block requires 2 cubes, in total 20. The between 10-30 vaos belong to the 1x2.
- L block requires 3 cubes, in total 30. The between 30-60 vaos belong to the L.
- 2x2 block requires 4 cubes, in total 40. The between 60-100 vaos belong to the 2x2.

Thus 100 cubes were needed to just draw the blocks. As for the lines that show the grids and for the pattern, 200 cubes are used. Additionally, to show the score 55 cubes were required to write the word ‘SCORE’ and an additional 45 cubes (15 each digit) to show the 3 digit score. All of the transformations, texture and light were done in the shaders.

To demonstrate the pattern behind, it was necessary to determine the cubes that should be colored differently. To achieve this, a 10x10 array was utilized, where the values -1 and 0 represented the absence or presence of a cube, respectively (see Appendix B). A similar approach was followed for the block array, which employed a 2x2 array. However, instead of 0, the actual vao index of the cubes was used. For example, if an L cube needed to be placed for the first time, the array would contain the values [{30, 31}, {32, -1}] (Appendix C). Appendix D provides details on the calculation of the initial Vao index for our randomly selected object.

Regarding the game area, a 10x10 array was also employed. Each index in the array corresponded to either the Vao index of the cube or -1 to indicate an empty spot. Consequently, in the display function, the cubes are drawn by iterating through the arrays and only rendering the present Vao indexes (refer to Appendix E). For the remaining elements, such as the grids and the pattern, a methodical approach was adopted. Since the number of cubes needed was already

known, for loops were employed to draw them individually, one by one. This allowed for a systematic rendering of the required elements in the game. (see Appendix F)

2.2. Transformation

To facilitate movement and rotation of the blocks, 4x4 transformation matrices are utilized. Additionally, animation is incorporated to enable smooth translation and rotation. It is important to note that different rotations are applied to the 1x1 and 1x2 blocks due to two limitations. The rotation for the 1x1 block is disallowed, whereas the 1x2 block is permitted to rotate 90 degrees clockwise only once.

Given that a 2x2 array is employed to determine the Vao indexes of the blocks (as explained in section 2.1), each rotation necessitates a modification of the array. For instance, let's consider the rotation of the L block mentioned above. The updated array would become [{31, -1}, {30, 32}].

As for collision detection, during each 1-unit descent, a comparison is made between the 2x2 block array and the corresponding 2x2 section of the 10x10 game area array. If the places that correspond after one unit fall are empty in the game area array, meaning the indexes have -1 as value, then there is no collision. If a collision is detected, it indicates that the block can no longer descend. In such cases, when the fall is finalized, the necessary adjustments are made to the game area array using the Vao index values of the falling block (see Appendix G, H, I).

2.3. Interactions

To select a cube within the block for removal, a picking method using the mouse was employed. To achieve this, each cube forming the block was colored differently in the back buffer. When the player clicks on a specific cube, the corresponding color pixel is identified, and as a result, the index value in the 2x2 block array is changed to -1, effectively making it empty. Similarly, the imitation of clicking on the "Play Again" button is accomplished using the picking technique, utilizing color rendering in the back buffer (refer to Appendix J).

2.4. Texture

The utilization of textures was incorporated into the game to apply various concepts covered throughout the class. At the end of the game, texture mapping was employed to cover the blocks with concept-based textures, allowing players to visualize the correct placements based on the reference image. To represent the 'house' pattern, a brick texture was utilized, while small red hearts were used for the 'Heart' pattern. In Figure 7 of Appendix E, it can be observed that incorrectly placed blocks are left untextured and appear as blue while correctly placed blocks have texture

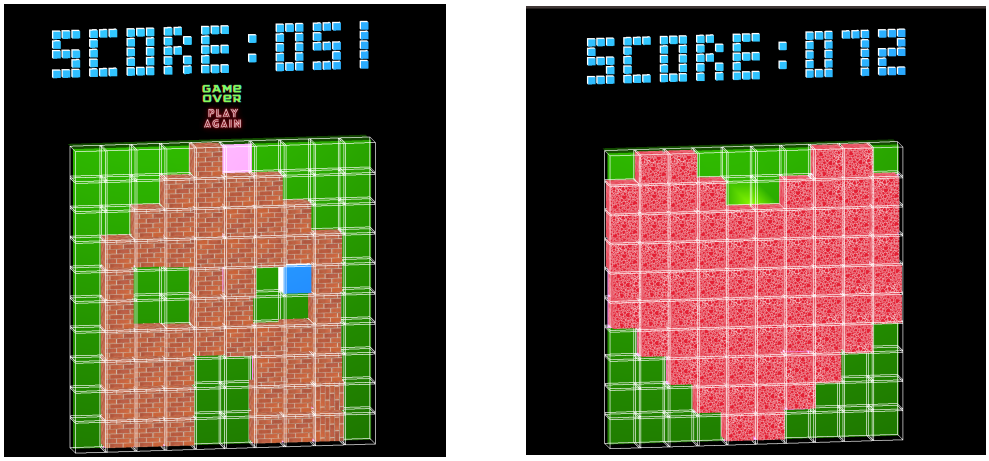

Figure 7 Two Patterns with Their Related Textures

2.5. Shading

Different shading options are used to improve visual experience. We wanted to have a bright setting with popping colors and to have a dark theme where there is a mysterious looking effect. They are implemented using two different shading options, Gouraud and Phong. Modified Phong illumination is also used. The player was allowed to change them any time in the game using 'S' key.

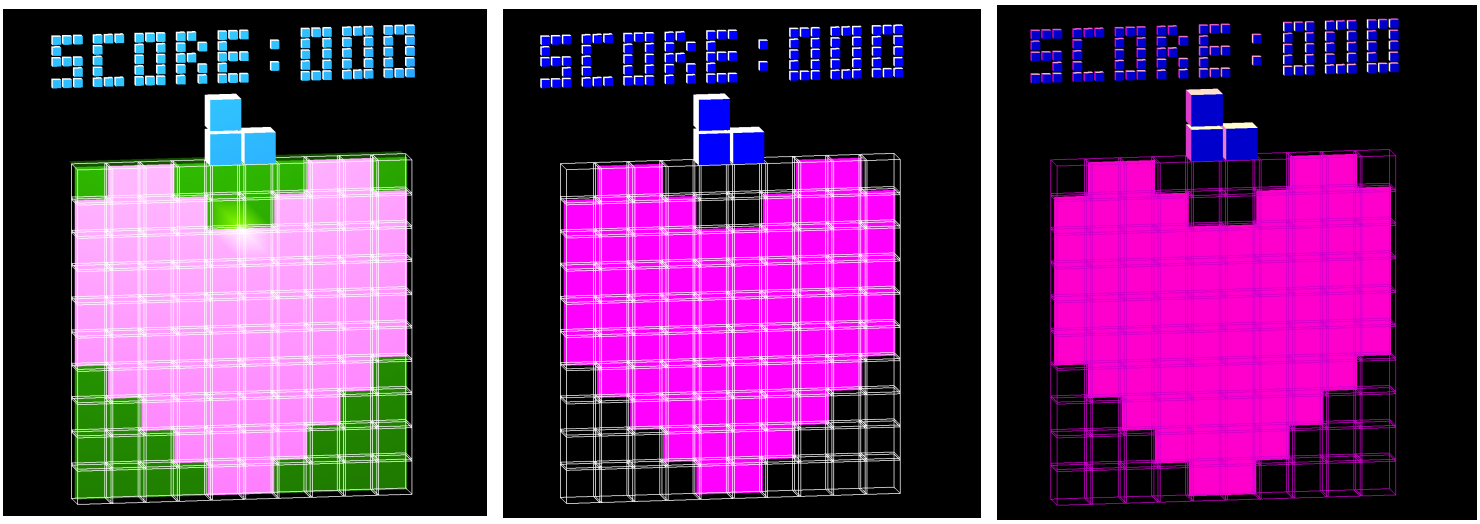

Figure 8 Different Themes

3. Tools

3.1. openGL

OpenGL (Open Graphics Library [1]) is an application program interface (API). It is used to render 2D and 3D vector graphics. It is supported on different platforms. OpenGL allows users to interact with a graphics processing unit (GPU) to accelerate rendering [2]. OpenGL API is used for the implementation of the game.

3.2. GLFW

GLFW is an library for OpenGL, OpenGL ES and Vulkan. It is open source, and supports Windows, macOS, X11 and Wayland. We used GLFW to create windows, surfaces, and context. Additionally, we utilize its functionalities to manage input events. The facilities can be implemented using GLFW API[3]. GLFW is used as an OpenGL library for implementation.

4. Summary

Kutris is a Tetris-like game where you are supposed to build object by following given patterns. By introducing the concept of following given patterns, Kutris add a unique mechanic to the traditional Tetris gameplay. Players are expected to analyze and recognize the given patters. According to that, they need to make changes on the randomly generated block and add them to the grid. This not only add a unique taste in to the game but also keeps the game exciting.

With the inclusion of patterns, Kutris creates a foundation for pattern based tetris game. The pool of patterns can be expanded regulary in the future to keep the game fresh. Also, a different scoring can be proposed to encourage players in finding different solutions.In this project we tried to apply all of the concepts learned through the course except Hierarchical Modeling as explained in section 2 in depth.

5. References

[1] OpenGL. [Online]. Available: https://www.opengl.org/.

[2] OpenGL - Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/OpenGL.

[3] GLFW. [Online]. Available: https://www.glfw.org

6. Appendix

```
// one by one 10*1
for(int i = 0; i < 10; i++){
    int j = 0 ;
    colorcube(i,j,oneByone);
}
// one by two 10*2
for(int i = 10; i < 30; i++){
    int j = i%2;
    colorcube(i,j,oneByTwo);
}
// L block 10*3
for(int i = 30; i < 60; i++){
    int j = i%3;
    colorcube(i,j,cubeL);
}
// two by two 10*4
for(int i = 60; i < 100; i++){
    int j = i%4;
    colorcube(i,j,twoBytwo);
}
```

Appendix A

```
// --------------------patterns ---------------------
int houseMatrix2[10][10] = {
        {-1,  0,  0,  0, -1, -1,  0,  0,  0, -1},
        {-1,  0,  0,  0, -1, -1,  0,  0,  0, -1},
        {-1,  0,  0,  0, -1, -1,  0,  0,  0, -1},
        {-1,  0,  0,  0,  0,  0,  0,  0,  0, -1},
        {-1,  0, -1, -1,  0,  0, -1, -1,  0, -1},
        {-1,  0, -1, -1,  0,  0, -1, -1,  0, -1},
        {-1,  0,  0,  0,  0,  0,  0,  0,  0, -1},
        {-1, -1,  0,  0,  0,  0,  0,  0, -1, -1},
        {-1, -1, -1,  0,  0,  0,  0, -1, -1, -1},
        {-1, -1, -1, -1,  0,  0, -1, -1, -1, -1}
};

int hearthMatrix2[10][10] = {
        {-1, -1, -1, -1,  0,  0, -1, -1, -1, -1},
        {-1, -1, -1,  0,  0,  0,  0, -1, -1, -1},
        {-1, -1,  0,  0,  0,  0,  0,  0, -1, -1},
        {-1,  0,  0,  0,  0,  0,  0,  0,  0, -1},
        { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
        { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
        { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
        { 0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
        { 0,  0,  0,  0, -1, -1,  0,  0,  0,  0},
        {-1,  0,  0, -1, -1, -1, -1,  0,  0, -1}
};
```

Appendix B

```
if (new_obj){
    for (int i = 0; i<2 ; i++){
        for (int j = 0; j<2 ; j++){
            cubeMatrix[i][j] = -1;
        }
    }
    if(blockToDraw == 1){
        cubeMatrix[0][0] = fromVao;
    }
    else if (blockToDraw == 2){
        cubeMatrix[0][0] = fromVao;
        cubeMatrix[0][1] = fromVao+1;
    }
    else if (blockToDraw == 3){
        cubeMatrix[0][0] = fromVao;
        cubeMatrix[1][0] = fromVao+1;
        cubeMatrix[0][1] = fromVao+2;
    }
    else if (blockToDraw == 4){
        cubeMatrix[0][0] = fromVao;
        cubeMatrix[1][0] = fromVao+1;
        cubeMatrix[1][1] = fromVao+2;
        cubeMatrix[0][1] = fromVao+3;
    }
    new_obj=false;
}
```

Appendix C

```
if (blockToDraw == onebyoneBlock) {
    fromVao = 10 - numberOfonebyoneBlock - 1;
}

else if (blockToDraw == onebytwoBlock) {
    fromVao = 10 + (10 - numberOfonebytwoBlock - 1) * 2 ;
}

else if (blockToDraw == LBlock) {
    fromVao = 10 + 20 + (10 - numberOfLBlock - 1) * 3 ;
}

else if (blockToDraw == twobytwoBlock) {
    fromVao = 10 + 20 + 30 + (10 - numberOftwobytwoBlock - 1) * 4 ;
}
```

Appendix D

```
//to draw the game area array (all of the blocks that are already been placed)
// and assign texture at the end of the game
        score = 0;
        for(int i = 0; i < sizeOfgame; i++){
            for (int j = 0; j<sizeOfgame; j++){
                if(gameMatrix[i][j] != -1 ){
                    if(gameEnded == 1 && patternToDraw == 0 && hearthMatrix2[i][j] != -1) {
                        glUniform1i(TextureOption, 1);
                    } else if (gameEnded == 1 && patternToDraw == 1 && houseMatrix2[i][j] != -1) {
                        glUniform1i(TextureOption, 1);
                    }
                    if(patternToDraw == 0 && hearthMatrix2[i][j] != -1){
                        score += 1;
                    } else if (patternToDraw == 0 && hearthMatrix2[i][j] == -1){
                        score -= 2;
                    }
                    if(patternToDraw == 1 && houseMatrix2[i][j] != -1){
                        score += 1;
                    } else if (patternToDraw == 1 && houseMatrix2[i][j] == -1){
                        score -= 2;
                    }
                    glBindVertexArray( vaoArray[gameMatrix[i][j]] );
                    glUniformMatrix4fv( ModelView, 1, GL_TRUE, Scale(1.5, 1.5, 1.5)*modelViews[gameMatrix[i][j]]);
                    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
                } else {
                    if(gameEnded == 1) {
                        glUniform1i(TextureOption, 0);
                    }
                }
            }
        }
        // update the score each time
        if(score < 0 ) score = 0;
        getScorePixel();
```

Appendix E

```
// ---------------draw the grid lines----------------------------
        glEnable(GL_LINE_SMOOTH);

            for(int a = 100; a<totalCubeRequired-(sizeOfgame*sizeOfgame); a++ ){
                glBindVertexArray( vaoArray[a] );
                glUniformMatrix4fv( ModelView, 1, GL_TRUE,  Scale(1.6, 1.6, 1.6)*modelViews[a]);
                glDrawArrays( GL_LINES, 0, 32 );
            }
// ---------------draw the randomly selected block ----------------------------
       for (int i = 0; i<2 ; i++){
            for (int j = 0; j<2 ; j++){
                if(cubeMatrix[i][j] != -1 && gameEnded != 1) {
                    glBindVertexArray( vaoArray[cubeMatrix[i][j]] );
                    glUniformMatrix4fv( ModelView, 1, GL_TRUE,  Scale(1.5, 1.5, 1.5)*modelViews[cubeMatrix[i][j]]);
                    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
                }
            }
        }
// ---------------draw the patterns----------------------------
            for(int i = totalCubeRequired-(sizeOfgame*sizeOfgame); i < totalCubeRequired; i++){
                glBindVertexArray( vaoArray[i] );
                glUniformMatrix4fv( ModelView, 1, GL_TRUE, Scale(1.6, 1.6, 1.6)*modelViews[i]);
                glDrawArrays( GL_TRIANGLES, 0, NumVertices );
            }
// ---------------draw the SCORE and 3 digits ----------------------------
        for(int i = totalCubeRequired; i < totalCubeRequired+scoreCube; i++){
            glBindVertexArray( vaoArray[i] );
            glUniformMatrix4fv( ModelView, 1, GL_TRUE, Scale(0.4, 0.4, 0.4)*modelViews[i]);
            glDrawArrays( GL_TRIANGLES, 0, NumVertices );
        }
```

Appendix F

```
// make the object fall while checking if there is collision
mat4 toTheBottom(mat4 model_view, int blockToDraw, int i){
    if (i == blockToDraw-1){
        move_degree+=0.2;
    }
    mat4 model_view_new;
    if(problem%4==0 )
        model_view_new =  model_view * Translate(0, -0.212, 0);
    if(problem%4==1)
        model_view_new =  model_view * Translate(0.212, 0, 0);
    if(problem%4==2)
        model_view_new =  model_view * Translate(0, 0.212, 0);
    if(problem%4==3)
        model_view_new =  model_view * Translate(-0.212, 0, 0);

    if(down_controller(controllerY) == true && move_degree == 1){
        move_degree=0;
        controllerY--;
    }
    else if(down_controller(controllerY) == false){
            go_down=false;
            move_degree=0;
            mess=0;
            updateGameMatrix();
        if(i == blockToDraw-1) forceFall = 1;
            return model_view;
    }

    return model_view_new;
}
```

```
// check collisions while making the block fall
bool down_controller(int controllerY){
    for (int i = 0; i<2 ; i++){
        for (int j = 0; j<2; j++){
            if(cubeMatrix[i][j] != -1){
                if(gameMatrix[controllerY+i-1][controllerX+j] != -1 && !(controllerY == sizeOfgame && i==1)){
                    return false;
                }
            }
        }
    }
    if(controllerY-1 == -1){
        return false;
    }
    return true;
}
```

Appendix G, H

```
void updateGameMatrix(){
    for (int i = 0; i<2; i++){
        for (int j = 0; j<2; j++){
            if(cubeMatrix[i][j] != -1){
                gameMatrix[controllerY+i][controllerX+j] = cubeMatrix[i][j];
            }
        }
    }
    vaoMatrix[totalNumberOfBlocks-1][0] = fromVao;
    vaoMatrix[totalNumberOfBlocks-1][1] = blockToDraw;


}
```

Appendix I

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if ( action == GLFW_PRESS && button == GLFW_MOUSE_BUTTON_LEFT) {
        glDrawBuffer(GL_BACK); //back buffer is default thus no need

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        if(!gameEnded){ //to assign different color to each of the cube that makes a block
            for (int i = 0; i<2 ; i++){
                for (int j = 0; j<2 ; j++){
                    pickingOption = (i*2)+j;
                    glUniform1i(UsePicking, pickingOption);
                    if(cubeMatrix[i][j] != -1) {
                        glBindVertexArray( vaoArray[cubeMatrix[i][j]] );
                        glUniformMatrix4fv( ModelView, 1, GL_TRUE, Scale(1.5, 1.5, 1.5)*modelViews[cubeMatrix[i][j]]);
                        glDrawArrays( GL_TRIANGLES, 0, NumVertices );
                    }
                }
            }
        }

        if (gameEnded){ // to color in the back buffer the PLAY AGAIN button
            for(int i = totalCubeRequired+scoreCube; i < totalCubeRequired+scoreCube+1; i++){
                glUniform1i(UsePicking, 0);
                glBindVertexArray( vaoArray[i] );
                glUniformMatrix4fv( ModelView, 1, GL_TRUE, Translate(0.37,1.73,1) * Scale(0.68, 0.68, 0.68)* RotateX( 0
                    ) * RotateY( 0 ) * RotateZ( 0.0 ));
                glDrawArrays( GL_TRIANGLES, 0, NumVertices );
            }
        }


        double x, y;
        glfwGetCursorPos(window, &x, &y);
```

Appendix J