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 pattenrs by strategically placing blocks.

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.

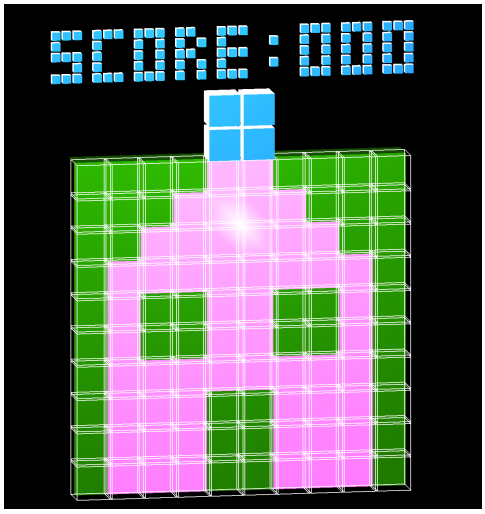

Figure 1 House Pattern

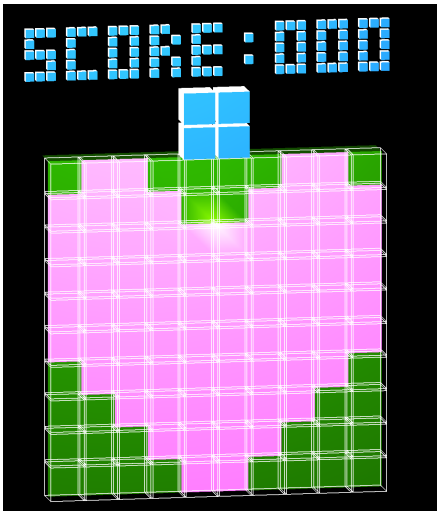

Figure 2 Heart Pattern

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.

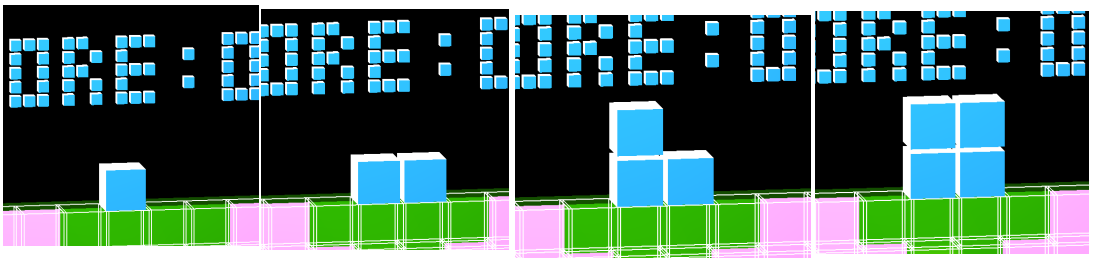

Figure 3 Differently Shaped Blocks

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)

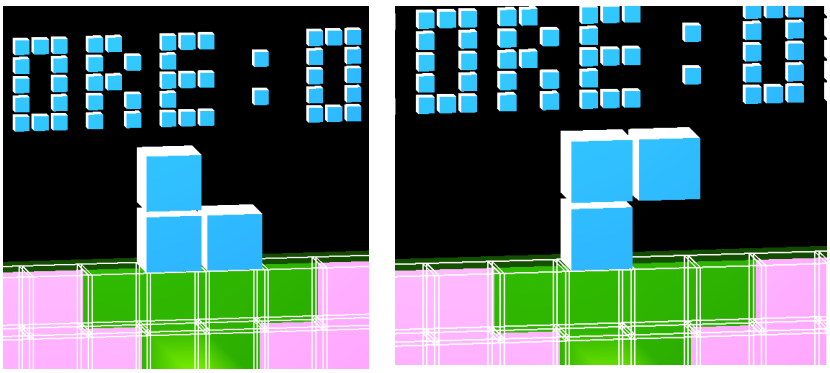

Figure 4 Rotating Blocks

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)

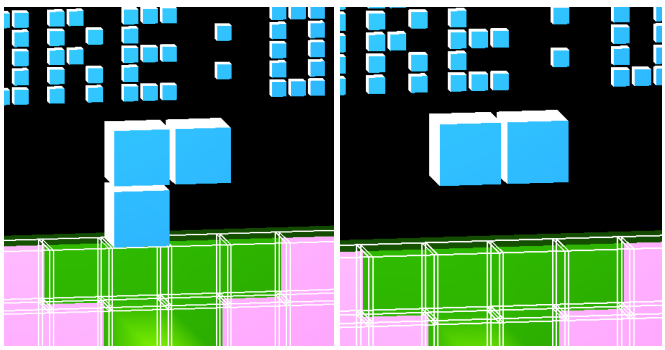

Figure 5 Deleting Cubes from Block

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)

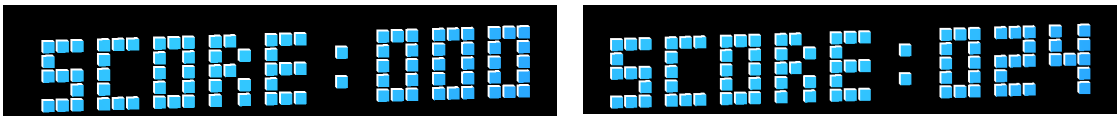

Figure 6 Score

2. Concepts

2.1. Basics

To draw objects to the scene, shader coding and 3D modeling is used. At the beginning of the game, we decided to place 10 of each block type to the buffer. We calculated the number of cube required to make the blocks 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 we needed 100 cubes to just draw the blocks. As for the lines that show the grids, we also used 100 cubes. Again for the showing pattern behind, we used 100 cubes. Additionally, to show the score we used 55 cubes to just write the word ‘SCORE’ and an additional 45 cubes (15 each digit) to show the 3 digit score. We did all of the transformations, texture and light in the shaders.

To show the pattern behind, we needed to determine which cubes to color differently. In order to do that we used a 10x10 array and -1 / 0 as values where -1 means there shouldn’t be a cube there, and 0 means there should be a one (see Appendix B). We also followed a similar pattern in the block array. In this case we used a 2x2 array, however, this time instead 0 we used the vao index of the cubes. For instance let’s assume we need to place an L cube and it is the first time we placed one, the array will have the values [{30, 31}, {32, -1}] (Appendix C). For the calculation of the from Vao which determines the initial voa index that our randomly selected object, see Appendix D. Lastly for the game area, we also used a 10x10 array. And for each index in the array, we used the vao index of the cube or -1 to indicate the spot is empty. Thus simply to draw the cubes in the display function, we iterate through the arrays and only draw the present vao indexes. (refer to Appendix E)

2.2. Transformation

To move and rotate the blocks, 4x4 transformation matrices are used. We also did animation of the translation and rotation. There were two limitations to using the same rotations for block 1x1 and 1x2. While we disallow the rotation for 1x1, we only allow one time rotation to 90 clockwise for 1x2 block. Since we used a 2x2 array to determine the vao indexes of the block that is drawn as explained in section 2.1, in each rotation we also needed to change the array. So let's assume we rotate the L block that is given as example above. The new array will become, [{31, -1}, {30, 32}]. As for the checking collisions, in each 1 unit fall we compared the 2x2 block array and the 2x2 part of the 10x10 game area array. If there will be a collision we assume that the block should no longer fall. When the collision happens we made the necessary changes to the game area array with the vao index values of the falling block. (see Appendix G, H, I)

2.3. Interactions

To select which cube in the block to remove, we used picking with the mouse. To do so, in the back buffer we colored each cube that constitutes the block in different colors. When the player clicks to a certain cube, where we understand it with the color pixel, we change the 2x2 block array value's index to -1(making it empty). Also, clicking on the Play Again button is imitated with picking where we used a coloring in the back buffer. (refer to Appendix J)

2.4. Texture

Since we wanted to use all of the concepts taught in the class, we used texture at the end of the game. We covered the blocks with concept-based texture to see the correct places of the reference image. For the 'house' pattern we used brick texture and for 'Heart' we used small red hearts. As can be seen in the Fig. 7 the wrong placed blocks are left to be blue without any textures. (Appendix E).

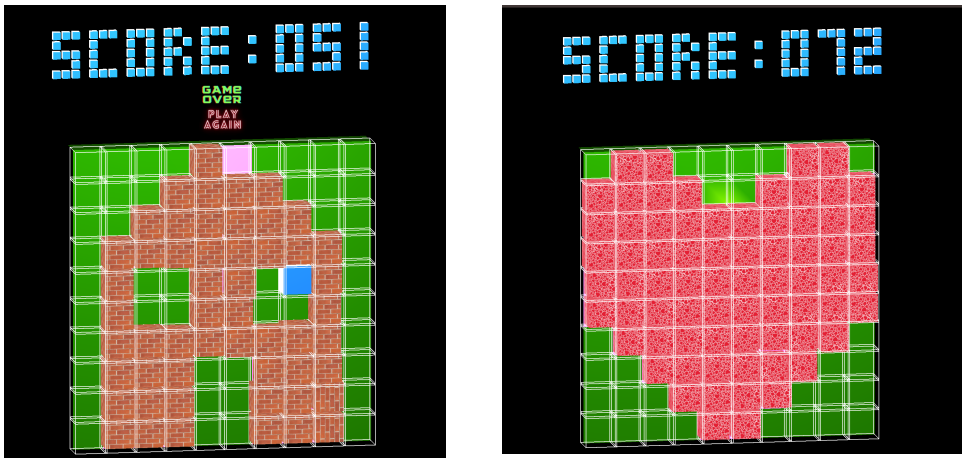

Figure 7 Two Patterns with Their Related Textures

2.5. Shading

Different shading options are used to improve visual experience. We wanted to have a bright setting with popping colors. Also we wanted 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. We allowed the user to change them any time in the game using ‘S’ key.

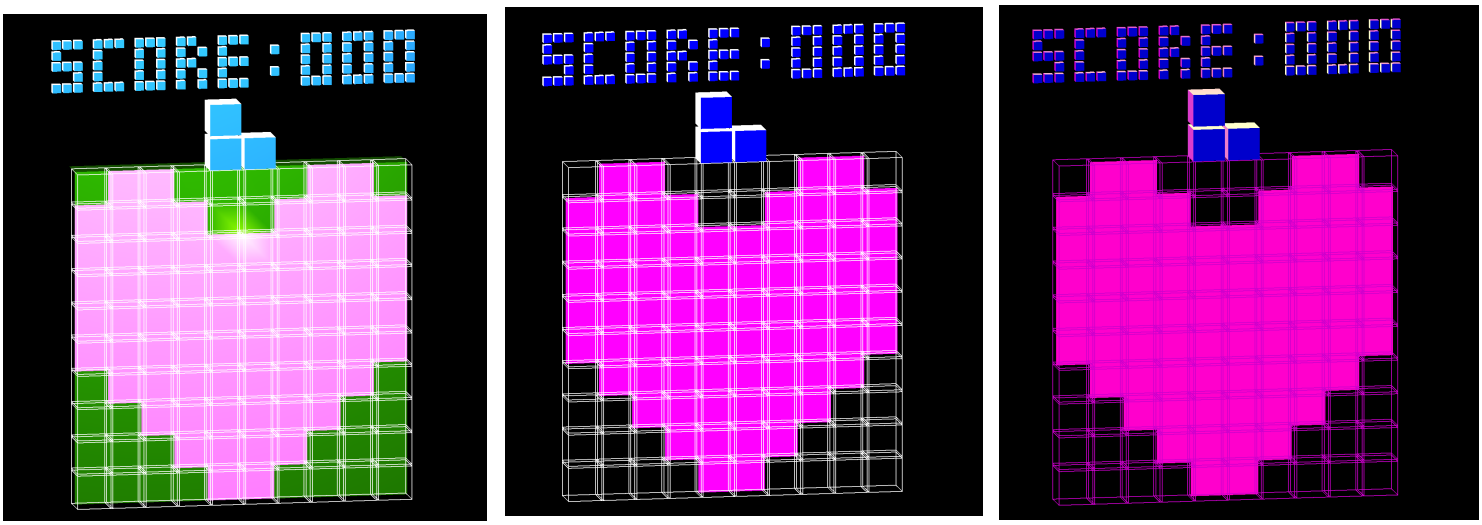

Figure 8 Different Themes

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