1 Programming Log

11/2/23

I spent 1 hour planning and designing the program. I decided on the 4 module structure and listed the new functions to create.

I then spent approximately 2 hours writing the matrix module and modifying the preexisting drawing module.

Finally, I spent approximately 4 hours writing the data and main modules. There were several small errors in the matrix and drawing modules which had to be fixed. Another significant challenge was figuring out proper settings for the window and camera in order to show the objects in a recognizable fashion.

2 Program Design

```
Split into four modules:
    matrix
        contains methods for basic matrix operations
    drawing
        contains all the methods for drawing on the screen
    data
        defines the specific objects being drawn
    main
        uses the methods of drawing to plot the objects of data

MATRIX
    typedef:
        vector4 -- 4 doubles, basic point/vector representation
        matrix4 -- 4x4 doubles, basic transform representation
    functions:
        DefineElementaryTransform(output, type, parameter)
            returns newly defined matrix4 for the transform
        BuildElementaryTransform(initial, type, parameter)
            updates initial to include specified transform
        MultiplyTransforms(left, right, output)
            multiplies matrices left and right, stores in output
        ApplyTransform(point, transform, output)
            multiplies vector point by transform matrix, stores in output
        CopyVector(source, destination)
            copies a vector from source to destination
        ApplyTransformInPlace(point, transform)
            stores in point
        CopyMatrix(source, destination)
            copies a matrix from source to destination
        ComposeTransform(initial, additional)
            multiples matrices, stores in initial

DRAWING
    much of this will be the same as the 2d assignment
    new functions:
        WriteName()
            writes name and class information on output
        Move3D(target)
        Draw3D(target)
            uses camera transform to turn input point to 2d
            then calls the 2d equivalent on the 2d projection
        DefineCameraTransform(focus, r, theta, phi, alpha)
            uses the Matrix functions to define (or re-define) the camera transform
```

```
        DrawShape(points, depth)
            draws from each point in points to the next, connects the last to the first
            if depth is nonzero, draws it as a second time, shifted on z axis
            connects corresponding points from the two drawings
        SetColor(r, g, b)
            sets the pen color

DATA
    DataFunction(x, y)
        returns z for the function at point (x, y)
    DataRubik(buffer)
        returns array of 9 arrays of vector4
        each array is the points for 1 square on the face of a rubik's cube
        buffer defines gap between squares
    DataRecognizer()
        returns array of arrays of vector4
        each array is one shape from the recognizer
    DataSpencer()
        returns array of arrays of vector4
        each array is a letter

MAIN
    main()
        draws the things
    DrawRubik(face, center)
        draws the stuff in face as one face of a cube centered at center
        changes color, rotates, draws the same stuff as another face
```

3 Source Code

3.1 main.cpp

```
//main.cpp
//Spencer Butler, 11/2/2023, CS324 3D Display

#include <stdio.h>

#include "matrix.h"
#include "drawing.h"
#include "data.h"

//Draws a cube with the contents of face on each face
//Each face has its own color
void DrawRubik(vector4 **face, int *sizes, int count, vector4 center);

#define PI 3.1415926
#define RAD(X) (X * PI * 1/180)

#define FUNCTION 1
#define CUBE_SOLID 2
#define CUBE_SPACED 3
#define CUBE_GRID 4
#define RECOGNIZER 5
#define NAME 6

static int cubeColors[6][3] = {
    {255, 0, 0},
    {0, 255, 0},
    {0, 0, 255},
    {255, 0, 255},
    {0, 255, 255},
    {255, 100, 0}
};

//draws things
int main() {
    int i, j, k, choice;
    InitGraphics();
    double vp[4] = {-1, 1, -1, 1};
    SetViewport(vp);

    printf("Select:\n");
    printf("\t 0 -- graph\n");
```

```
printf("\t 1 -- rubik's cube solid\n");
printf("\t 2 -- rubik's cube gaps\n");
printf("\t 3 -- many rubik's cubes\n");
printf("\t 4 -- recognizer\n");
printf("\t 5 -- name in block letters \n");

if (scanf_s("%d", &choice) != 1) {
    printf("Error reading drawing choice.\n");
    return 1;
}


//graph
if(choice == 0) {
    double w[4] = {-8, 8, -8, 8};
    SetWindow(w);
    vector4 focalPoint;
    focalPoint[0] = 0;
    focalPoint[1] = 0;
    focalPoint[2] = 0.5;
    focalPoint[3] = 1;
    DefineCameraTransform(focalPoint, 25, RAD(25), RAD(-70), RAD(0));

    vector4 p;
    p[0] = 0;
    p[1] = 0;
    p[2] = 0;
    p[3] = 1;

    int steps = 50;

    //draw all lines varying y
    for(p[0] = -2 * PI; p[0] <= 2 * PI; p[0] = p[0] + (4 * PI / steps)) {
        p[1] = -2 * PI;
        p[2] = DataFunction(p[0], p[1]);
        Move3D(p);
        for(; p[1] <= 2 * PI; p[1] = p[1] + (4 * PI / steps)) {
            p[2] = DataFunction(p[0], p[1]);
            Draw3D(p);
        }
    }

    //draw all lines varying x
    for(p[1] = -2 * PI; p[1] <= 2 * PI; p[1] = p[1] + (4 * PI / steps)) {
        p[0] = -2 * PI;
```

```
        p[2] = DataFunction(p[0], p[1]);
        Move3D(p);
        for(; p[0] <= 2 * PI; p[0] = p[0] + (4 * PI / steps)) {
            p[2] = DataFunction(p[0], p[1]);
            Draw3D(p);
        }
    }
}

//single rubik
if(choice == 1 || choice == 2) {
    double w[4] = {-1.5, 1.5, -1.5, 1.5};
    SetWindow(w);
    vector4 focalPoint;
    focalPoint[0] = 0;
    focalPoint[1] = 0;
    focalPoint[2] = 0.5;
    focalPoint[3] = 1;
    DefineCameraTransform(focalPoint, 25, RAD(20), RAD(-70), RAD(0));

    vector4 origin;
    origin[0] = 0;
    origin[1] = 0;
    origin[2] = 0;
    origin[3] = 1;

    vector4 squares[9][4];
    int fours[9];
    for(i = 0; i < 9; i++) { fours[i] = 4; }
    if(choice == 1) {
        DataRubik(0.0, squares);
    } else {
        DataRubik(0.2, squares);
    }

    vector4 *sq[9];
    for(i = 0; i < 9; i++) {
        sq[i] = squares[i];
    }
    DrawRubik(sq, fours, 9, origin);
}

//many rubik
if(choice == 3) {
    double w[4] = {-4.5, 20, -1.5, 20};
```

```
    SetWindow(w);
    vector4 focalPoint;
    focalPoint[0] = 0;
    focalPoint[1] = 0;
    focalPoint[2] = 0.5;
    focalPoint[3] = 1;
    DefineCameraTransform(focalPoint, 30, RAD(20), RAD(-70), RAD(0));

    vector4 origin;
    origin[0] = 0;
    origin[1] = 0;
    origin[2] = 0;
    origin[3] = 1;

    vector4 squares[9][4];
    int fours[9];
    for(i = 0; i < 9; i++) { fours[i] = 4; }

    vector4 *sq[9];
    for(i = 0; i < 9; i++) {
        sq[i] = squares[i];
    }
    for(i = 0; i < 8; i++) {
        origin[1] = 0;
        for(j = 0; j < 8; j++) {
            origin[2] = 0;
            for(k = 0; k < 8; k++) {
                DataRubik(0.2, squares);
                DrawRubik(sq, fours, 9, origin);
                origin[2] = origin[2] + 1.5;
            }
            origin[1] = origin[1] + 1.5;
        }
        origin[0] = origin[0] + 1.5;
    }

}

//recognizer
if(choice == 4) {
    double w[4] = {-1.5, 1.5, -1.5, 1.5};
    SetWindow(w);
    vector4 focalPoint;
    focalPoint[0] = 0;
    focalPoint[1] = 0;
```

```
        focalPoint[2] = 0.5;
        focalPoint[3] = 1;

        DefineCameraTransform(focalPoint, 25, RAD(20), RAD(-75), RAD(0));

        vector4 **shapes;
        int *sizes;
        int count;
        SetColor(128, 0, 64);
        DataRecognizer(&shapes, &sizes, &count);
        for(i = 0; i < count; i++) {
            DrawShape(shapes[i], sizes[i], 0.15, 1);
        }
    }

    //spencer
    if(choice == 5) {
        double w[4] = {-1.0, 8.0, -1.5, 1.5};
        SetWindow(w);
        vector4 focalPoint;
        focalPoint[0] = 0.5;
        focalPoint[1] = 0;
        focalPoint[2] = 0.5;
        focalPoint[3] = 1;

        DefineCameraTransform(focalPoint, 20, RAD(5), RAD(-70), RAD(0));

        vector4 *shapes[7];
        int sizes[7];
        DataSpencer(shapes, sizes);
        for(i = 0; i < 7; i++) {
            DrawShape(shapes[i], sizes[i], 0.5, 1);
        }
    }

    return EventLoop();
}

//Draws a cube with the contents of face on each face
//Each face has its own color
void DrawRubik(vector4 **face, int *sizes, int count, vector4 center) {
    int i, j, k;
    int colIndex = 0;
    int *color;
```

```
matrix4 rotationMatrix;
DefineElementaryTransform(&rotationMatrix, ROTATE_X, PI / 2);

matrix4 translationMatrix;
DefineElementaryTransform(&translationMatrix, TRANSLATE_X, center[0]);
BuildElementaryTransform(&translationMatrix, TRANSLATE_Y, center[1]);
BuildElementaryTransform(&translationMatrix, TRANSLATE_Z, center[2]);

matrix4 reverseTranslationMatrix;
DefineElementaryTransform(&reverseTranslationMatrix, TRANSLATE_X, -1 * center[0]);
BuildElementaryTransform(&reverseTranslationMatrix, TRANSLATE_Y, -1 * center[1]);
BuildElementaryTransform(&reverseTranslationMatrix, TRANSLATE_Z, -1 * center[2]);

//four easy faces
for(i = 0; i < 4; i++) {
    //translate each point
    for(j = 0; j < count; j++) {
        for(k = 0; k < sizes[j]; k++) {
            ApplyTransformInPlace(&(face[j][k]), translationMatrix);
        }
    }

    //draw this face
    color = cubeColors[colIndex];
    SetColor(color[0], color[1], color[2]);
    for(j = 0; j < count; j++) {
        DrawShape(face[j], sizes[j], 0, 0);
    }

    //rotate each point
    for(j = 0; j < count; j++) {
        for(k = 0; k < sizes[j]; k++) {
            ApplyTransformInPlace(&(face[j][k]), reverseTranslationMatrix);
            ApplyTransformInPlace(&(face[j][k]), rotationMatrix);
        }
    }
    colIndex++;
}

//draw remaining side 1
DefineElementaryTransform(&rotationMatrix, ROTATE_Y, PI / 2);
//rotate then translate each point
for(j = 0; j < count; j++) {
    for(k = 0; k < sizes[j]; k++) {
        ApplyTransformInPlace(&(face[j][k]), rotationMatrix);
```

```
            ApplyTransformInPlace(&(face[j][k]), translationMatrix);
        }
    }
    color = cubeColors[colIndex];
    SetColor(color[0], color[1], color[2]);
    for(j = 0; j < count; j++) {
        DrawShape(face[j], sizes[j], 0, 0);
    }

    colIndex++;
    //draw remaining side 2
    //rotate then translate each point
    for(j = 0; j < count; j++) {
        for(k = 0; k < sizes[j]; k++) {
            ApplyTransformInPlace(&(face[j][k]), reverseTranslationMatrix);
            ApplyTransformInPlace(&(face[j][k]), rotationMatrix);
            ApplyTransformInPlace(&(face[j][k]), rotationMatrix);
            ApplyTransformInPlace(&(face[j][k]), translationMatrix);
        }
    }
    color = cubeColors[colIndex];
    SetColor(color[0], color[1], color[2]);
    for(j = 0; j < count; j++) {
        DrawShape(face[j], sizes[j], 0, 0);
    }
}
```

3.2 drawing.h

```
//drawing.h
//Spencer Butler, 11/2/2023, CS324 3D Display

#ifndef drawing_h
#define drawing_h

#include "matrix.h"

//Initializes the SFML window and coordinate systems for drawing
void InitGraphics();

//Converts a point from window to viewport coordinates
void WindowToViewPort(double before[2], double after[2]);
```

```
//Converts a point from screenspace coordinates to drawing-window coordinates (pixels)
void ScreenToDevice(double before[2], double after[2]);

//Moves the drawing cursor to target without drawing anything
void MoveTo2D(double target[2]);

//Draws a line to target, moving the cursor
void DrawTo2D(double target[2]);

//Projects target according to the camera transform, MoveTo2D(results)
void Move3D(vector4 target);

//Projects target according to the camera transform, DrawTo2D(results)
void Draw3D(vector4 target);

//Draws the closed shape specified by points
//If depth is nonzero, extrudes shape along the chosen axis
void DrawShape(vector4 points[], int size, double depth, int axis);

//Sets the pen color
void SetColor(int r, int g, int b);

//Sets viewport coordinates to given bounds (x, x, y, y)
void SetViewport(double bounds[4]);

//Sets window coordinates to given bounds (x, x, y, y)
void SetWindow(double bounds[4]);

//Defines the camera as pointing at focus
//Camera location defined via spherical coords (r, theta, phi) and orientation alpha
void DefineCameraTransform(vector4 focus, double r, double theta, double phi, double alp

//Returns 1 if point p is within region r
int within(double r[4], double p[2]);

//Calculates midpoint of points p and q, stores in r
void midpoint(double p[2], double q[2], double r[2]);

//Linearly interpolates val, starting within startRange, and ending within endRange
double lerp(double startRange[2], double val, double endRange[2]);

//Swaps *low and *high if they're out of order
void order(double *low, double *high);
```

```
//Returns val constrained to within [floor, ceil]
int constrain(int floor, int val, int ceil);

//Writes name and class information on output
void writeName();

//Loops for the lifetime of an SFML render window
int EventLoop();

#endif
```

3.3 drawing.cpp

```
//drawing.cpp
//Spencer Butler, 11/2/2023, CS324 3D Display

#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include <stdio.h>

#include "drawing.h"

static int device[2];
static double window[4];
static double viewport[4];

static double cursor[2];
static sf::RenderWindow *rw;
static matrix4 camera;
static sf::Color penColor(0, 0, 0, 255);


//Initializes the SFML window and coordinate systems for drawing
void InitGraphics() {
    device[0] = 800;
    device[1] = 800;

    rw = new sf::RenderWindow(sf::VideoMode(device[0], device[1]), "3D Display", sf::Sty
    rw->clear(sf::Color::White);

    double wtemp[4] = {-5, 5, -5, 5};
    SetWindow(wtemp);

    double vtemp[4] = {-1, 1, -1, 1};
```

```
    SetViewport(vtemp);
}

//Converts a point from window to viewport coordinates
void WindowToViewPort(double before[2], double after[2]) {
    after[0] = lerp(window, before[0], viewport);
    after[1] = lerp(window + 2, before[1], viewport + 2);
}

//Converts a point from screenspace coordinates to drawing-window coordinates (pixels)
void ScreenToDevice(double before[2], double after[2]) {
    double screen[4] = {-1, 1, -1, 1};
    double dev[4] = {0, device[0], 0, device[1]};

    after[0] = (int) lerp(screen, before[0], dev);
    //do device[1] - lerp() to convert from 0-bottom to 0-top
    after[1] = (int) device[1] - lerp(screen + 2, before[1], dev + 2);
}

//Moves the drawing cursor to target without drawing anything
void MoveTo2D(double target[2]) {
    cursor[0] = target[0];
    cursor[1] = target[1];
}

//Draws a line to target, moving the cursor
void DrawTo2D(double target[2]) {
    double temp[2];
    double startPixels[2];
    double endPixels[2];

    //check if start+end+midpoint are all outside the window
    //if they all are, this line normally shouldn't be drawn
    //culling these lines prevents bleedthrough along the window's edges
    midpoint(cursor, target, temp);
    if(!(within(window, cursor) || within(window, temp) || within(window, target))) {
        MoveTo2D(target);
        return;
    }

    WindowToViewPort(cursor, temp);
    ScreenToDevice(temp, startPixels);

    WindowToViewPort(target, temp);
    ScreenToDevice(temp, endPixels);
```

```
    sf::Vertex vertices[2] = {
        sf::Vertex(sf::Vector2f(startPixels[0], startPixels[1]), penColor),
        sf::Vertex(sf::Vector2f(endPixels[0], endPixels[1]), penColor)
    };

    rw->draw(vertices, 2, sf::Lines);

    MoveTo2D(target);
}

//Projects target according to the camera transform, MoveTo2D(results)
void Move3D(vector4 target) {
    vector4 after;
    ApplyTransform(target, camera, &after);
    //double point[2] = {after[0] / after[3], after[1] / after[3]};
    MoveTo2D(&(after[0]));
}

//Projects target according to the camera transform, DrawTo2D(results)
void Draw3D(vector4 target) {
    vector4 after;
    ApplyTransform(target, camera, &after);
    DrawTo2D(&(after[0]));
}

//Draws the closed shape specified by points
//If depth is nonzero, extrudes shape along the chosen axis
void DrawShape(vector4 points[], int size, double depth, int axis) {
    int i;
    vector4 deepPoint;
    matrix4 depthTransform;
    Move3D(points[0]);
    for(i = 0; i < size; i++) {
        Draw3D(points[i]);
    }
    Draw3D(points[0]);

    if(depth > 0.01 || depth < 0.01) {
        DefineElementaryTransform(&depthTransform, TRANSLATE + axis, depth);

        ApplyTransform(points[0], depthTransform, &deepPoint);
        Move3D(deepPoint);
        for(i = 0; i < size; i++) {
```

```
            //draw to next point on deep face
            ApplyTransform(points[i], depthTransform, &deepPoint);
            Draw3D(deepPoint);

            //draw connection to non-deep point
            Move3D(points[i]);
            Draw3D(deepPoint);
        }
        ApplyTransform(points[0], depthTransform, &deepPoint);
        Draw3D(deepPoint);
    }

}

//Sets the pen color
void SetColor(int r, int g, int b) {
    penColor.r = constrain(0, r, 255);
    penColor.g = constrain(0, g, 255);
    penColor.b = constrain(0, b, 255);
}

//Sets viewport coordinates to given bounds (x, x, y, y)
void SetViewport(double bounds[4]) {
    int i;
    order(bounds, bounds + 1);
    order(bounds + 2, bounds + 3);
    for(i = 0; i < 4; i++) {
        if(bounds[i] > 1) {
            viewport[i] = 1;
        } else if (bounds[i] < -1) {
            viewport[i] = -1;
        } else {
            viewport[i] = bounds[i];
        }
    }
}

//Sets window coordinates to given bounds (x, x, y, y)
void SetWindow(double bounds[4]) {
    int i;
    order(bounds, bounds + 1);
    order(bounds + 2, bounds + 3);
    for(i = 0; i < 4; i++) {
        window[i] = bounds[i];
    }
```

```
}

//Defines the camera as pointing at focus
//Camera location defined via spherical coords (r, theta, phi) and orientation alpha
void DefineCameraTransform(vector4 focus, double r, double theta, double phi, double alp
    DefineElementaryTransform(&camera, TRANSLATE_X, -1 * focus[0]);
    BuildElementaryTransform(&camera, TRANSLATE_Y, -1 * focus[1]);
    BuildElementaryTransform(&camera, TRANSLATE_Z, -1 * focus[2]);

    BuildElementaryTransform(&camera, ROTATE_Y, -1 * theta);
    BuildElementaryTransform(&camera, ROTATE_X, phi);
    BuildElementaryTransform(&camera, ROTATE_Z, -1 * alpha);

    BuildElementaryTransform(&camera, PERSPECTIVE, r);
}

//Returns 1 if point p is within region r
int within(double r[4], double p[2]) {
    //x is out of bounds
    if(p[0] < fmin(r[0], r[1]) || p[0] > fmax(r[0], r[1])) {
        return 0;
    }

    //y is out of bounds
    if(p[1] < fmin(r[2], r[3]) || p[1] > fmax(r[2], r[3])) {
        return 0;
    }

    return 1;
}

//Calculates midpoint of points p and q, stores in r
void midpoint(double p[2], double q[2], double r[2]) {
    r[0] = (p[0] + q[0]) * 0.5;
    r[1] = (p[1] + q[1]) * 0.5;
}

//Linearly interpolates val, starting within startRange, and ending within endRange
double lerp(double startRange[2], double val, double endRange[2]) {
    order(startRange, startRange + 1);
    order(endRange, endRange + 1);

    if(val < startRange[0]) {
        val = startRange[0];
    } else if(val > startRange[1]) {
```

```
        val = startRange[1];
    }

    //map to [0, 1]
    val = (val - startRange[0]) / (startRange[1] - startRange[0]);

    //map to endRange
    val = (val * (endRange[1] - endRange[0])) + endRange[0];
    return val;
}

//Swaps *low and *high if they're out of order
void order(double *low, double *high) {
    double i;
    if(*low > *high) {
        i = *low;
        *low = *high;
        *high = i;
    }
}

//Returns val constrained to within [floor, ceil]
int constrain(int floor, int val, int ceil) {
    if(val < floor)
        return floor;
    if(val > ceil)
        return ceil;
    return val;
}

//Writes name and class information on output
void writeName() {
    sf::Font font;
    if (!font.loadFromFile("C:\\WINDOWS\\FONTS\\TIMES.TTF")) {
        printf("Error loading font.\n");
        return;
    }
    sf::Text text("Spencer Butler, CS324 Fall 2023", font, 20);
    text.setFillColor(sf::Color::Black);
    text.setPosition(25, 35);
    rw->draw(text);
}

//Loops for the lifetime of an SFML render window
int EventLoop() {
```

```
    writeName();
    sf::Event event;
    rw->display();
    while (rw->isOpen()) {
        while (rw->pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                rw->close();
            }
        }
    }
    return 0;
}
```

3.4 matrix.h

```
//matrix.h
//Spencer Butler, 11/2/2023, CS324 3D Display

#ifndef matrix_h
#define matrix_h
typedef double vector4[4];
typedef double matrix4[4][4];

#define IDENTITY 0

#define TRANSLATE 1
#define TRANSLATE_X 1
#define TRANSLATE_Y 2
#define TRANSLATE_Z 3

#define ROTATE 4
#define ROTATE_X 4
#define ROTATE_Y 5
#define ROTATE_Z 6

#define PERSPECTIVE 7

//Multiplies point by transform, stores result in output
void ApplyTransform(vector4 point, matrix4 transform, vector4 *output);

//Copies the vector at source to destination
void CopyVector(vector4 source, vector4 *destination);

//Multiplies point by transform, stores result in point
```

```
void ApplyTransformInPlace(vector4 *point, matrix4 transform);

//Multiplies left by right, stores result in output
void MultiplyTransforms(matrix4 left, matrix4 right, matrix4 *output);

//Copies the matrix at source to destination
void CopyMatrix(matrix4 source, matrix4 *destination);

//Multiplies initial by additional and stores the results in initial
void ComposeTransform(matrix4 *initial, matrix4 additional);

//Stores a new matrix4 representing the specified transform in output
void DefineElementaryTransform(matrix4 *output, int type, double parameter);

//Modifies initial to also apply the specified transform
void BuildElementaryTransform(matrix4 *initial, int type, double parameter);

//Prints the vector
void printVector(vector4 v);

//Prints the matrix
void printMatrix(matrix4 m);

#endif
```

3.5 matrix.cpp

```
//matrix.cpp
//Spencer Butler, 11/2/2023, CS324 3D Display

#include <math.h>
#include <stdio.h>

#include "matrix.h"

//Multiplies point by transform, stores result in output
void ApplyTransform(vector4 point, matrix4 transform, vector4 *output) {
    int i, j;
    for(i = 0; i < 4; i++) {
        (*output)[i] = 0;
        for(j = 0; j < 4; j++) {
```

```
            (*output)[i] = (*output)[i] + (point[j] * transform[j][i]);
        }
    }
    for(i = 0; i < 4; i++) {
        (*output)[i] = (*output)[i] / (*output)[3];
    }
}

//Copies the vector at source to destination
void CopyVector(vector4 source, vector4 *destination) {
    int i;
    for(i = 0; i < 4; i++) {
        (*destination)[i] = source[i];
    }
}

//Multiplies point by transform, stores result in point
void ApplyTransformInPlace(vector4 *point, matrix4 transform) {
    vector4 proxy;
    ApplyTransform(*point, transform, &proxy);
    CopyVector(proxy, point);
}

//Multiplies left by right, stores result in output
void MultiplyTransforms(matrix4 left, matrix4 right, matrix4 *output) {
    int i, j, k;
    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            (*output)[i][j] = 0;
            for(k = 0; k < 4; k++) {
                (*output)[i][j] = (*output)[i][j] + (left[i][k] * right[k][j]);
            }
        }
    }
}

//Copies the matrix at source to destination
void CopyMatrix(matrix4 source, matrix4 *destination) {
    int i, j;
    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            (*destination)[i][j] = source[i][j];
        }
    }
}
```

```
//Multiplies initial by additional and stores the results in initial
void ComposeTransform(matrix4 *initial, matrix4 additional) {
    matrix4 proxy;
    MultiplyTransforms(*initial, additional, &proxy);
    CopyMatrix(proxy, initial);
}

//Returns a new matrix4 representing the specified transform
void DefineElementaryTransform(matrix4 *output, int type, double parameter) {
    int i, j;
    //initialize with identity matrix
    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            (*output)[i][j] = (i == j);
        }
    }

    switch(type) {
    case TRANSLATE_X:
        (*output)[3][0] = parameter;
        break;
    case TRANSLATE_Y:
        (*output)[3][1] = parameter;
        break;
    case TRANSLATE_Z:
        (*output)[3][2] = parameter;
        break;

    case ROTATE_X:
        (*output)[1][1] = cos(parameter);
        (*output)[1][2] = sin(parameter);
        (*output)[2][1] = -1 * sin(parameter);
        (*output)[2][2] = cos(parameter);
        break;
    case ROTATE_Y:
        (*output)[0][0] = cos(parameter);
        (*output)[0][2] = -1 * sin(parameter);
        (*output)[2][0] = sin(parameter);
        (*output)[2][2] = cos(parameter);
        break;
    case ROTATE_Z:
        (*output)[0][0] = cos(parameter);
        (*output)[0][1] = sin(parameter);
        (*output)[1][0] = -1 * sin(parameter);
```

```
        (*output)[1][1] = cos(parameter);
        break;

    case PERSPECTIVE:
        (*output)[2][3] = -1 / parameter;
        break;

    //catches type == identity, and all unexpected entries
    default:
        break;
    }
}

//Modifies initial to also apply the specified transform
void BuildElementaryTransform(matrix4 *initial, int type, double parameter) {
    matrix4 additional;
    DefineElementaryTransform(&additional, type, parameter);
    ComposeTransform(initial, additional);
}

//Prints the vector
void printVector(vector4 v) {
    printf("(%lf, %lf, %lf, %lf)", v[0], v[1], v[2], v[3]);
}

//Prints the matrix
void printMatrix(matrix4 m) {
    int i, j;
    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            printf("%lf ", m[i][j]);
        }
        printf("\n");
    }
}
```

3.6 data.h

```
//data.h
//Spencer Butler, 11/2/2023, CS324 3D Display

#ifndef data_h
```

```
#define data_h

#include "matrix.h"

//Returns the value of a specific function at (x, y)
double DataFunction(double x, double y);

//Generates shapes representing the top face of a rubik's cube centered at (0, 0, 0)
//Buffer controls space between adjacent squares
//Stores result in output
void DataRubik(double buffer, vector4 output[9][4]);

//Generates shapes representing a Tron recognizer
//Number of shapes is stored in count
//Number of points in each shape is stored in sizes
//Shapes are stored in output
void DataRecognizer(vector4 ***output, int **sizes, int *count);

//Generates shapes representing the word Spencer
//Returns total of 7 shapes
//Number of points in each shape is stored in sizes
void DataSpencer(vector4 *output[7], int sizes[7]);

#endif
```

3.7 data.cpp

```
//data.cpp
//Spencer Butler, 11/2/2023, CS324 3D Display

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define PI 3.1415926

#include "matrix.h"
#include "data.h"

//Returns the value of a specific function at (x, y)
double DataFunction(double x, double y) {
    double r, numer, denom;
    //r = sqrt(x * x + y * y);
```

```
    r = x * x + y * y;
    numer = sin(r) / r;
    denom = 9 * cos(x / (y + 0.02));
    return numer / denom;
}

//Generates shapes representing the top face of a rubik's cube centered at (0, 0, 0)
//Buffer controls space between adjacent squares
//Stores result in output
void DataRubik(double buffer, vector4 output[9][4]) {
    int i, j, k;
    vector4 point;
    matrix4 translationMatrix;
    matrix4 rotationMatrix;
    DefineElementaryTransform(&rotationMatrix, ROTATE_Z, PI / 2);
    double sideLength = (1.0 / 3.0) * (1 - buffer);

    //set point to be topleft of single square centered at (0, 0, 1)
    point[0] = -1 * (sideLength * 0.5);
    point[1] = -1 * (sideLength * 0.5);
    point[2] = 0.5;
    point[3] = 1.0;

    //initialize translationMatrix for center of the upper-left square
    DefineElementaryTransform(&translationMatrix, TRANSLATE_X, (-1.0 / 2.0) + (1.0 / 6.0
    BuildElementaryTransform(&translationMatrix, TRANSLATE_Y, (-1.0 / 2.0) + (1.0 / 6.0)

    //grid of 3x3 squares
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 3; j++) {
            //each square made of 4 points
            for(k = 0; k < 4; k++) {
                //translate current corner to the current square, store result in output
                ApplyTransform(point, translationMatrix, &(output[(3 * i) + j][k]));
                //rotate to next corner
                ApplyTransformInPlace(&point, rotationMatrix);
            }
            //move 1/3 over, to center of next square
            BuildElementaryTransform(&translationMatrix, TRANSLATE_X, (1.0 / 3.0));
        }
        //move 1/3 down and 3/3 left, to center of first square on next row
        BuildElementaryTransform(&translationMatrix, TRANSLATE_Y, (1.0 / 3.0));
        BuildElementaryTransform(&translationMatrix, TRANSLATE_X, -1.0);
    }
}
```

```
//Generates shapes representing a Tron recognizer
//Number of shapes is stored in count
//Number of points in each shape is stored in sizes
//Shapes are stored in output
void DataRecognizer(vector4 ***output, int **sizes, int *count) {
    int i, j;
    vector4 point;
    point[0] = -1;
    point[1] = 0;
    point[2] = 0;
    point[3] = 1;

    *count = 8;
    (*sizes) = (int *) malloc(sizeof(int) * (*count));
    (*output) = (vector4 **) malloc(sizeof(vector4 *) * (*count));

    (*sizes)[0] = 5;
    (*sizes)[1] = 4;
    (*sizes)[2] = 10;
    for(i = 0; i < 3; i++) {
        (*sizes)[i + 3] = (*sizes)[i];
    }
    (*sizes)[6] = 4;
    (*sizes)[7] = 4;
    for(i = 0; i < 8; i++) {
        (*output)[i] = (vector4 *) malloc(sizeof(vector4) * ((*sizes)[i]));
    }

    //draw little column with trapezoid base
    {
        CopyVector(point, (*output)[0] + 0);

        point[0] = -0.7;
        CopyVector(point, (*output)[0] + 1);

        point[0] = -0.8;
        point[2] = 0.1;
        CopyVector(point, (*output)[0] + 2);

        point[2] = 0.5;
        CopyVector(point, (*output)[0] + 3);

        point[0] = -1;
        CopyVector(point, (*output)[0] + 4);
```

```
}

//draw square above that
{
    point[2] = 0.7;
    CopyVector(point, (*output)[1] + 0);

    point[0] = -0.8;
    CopyVector(point, (*output)[1] + 1);

    point[2] = 0.9;
    CopyVector(point, (*output)[1] + 2);

    point[0] = -1.0;
    CopyVector(point, (*output)[1] + 3);
}

//draw forkthing above that
{
    point[0] = -1.1;
    point[2] = 1.0;
    CopyVector(point, (*output)[2] + 0);

    point[0] = -0.75;
    CopyVector(point, (*output)[2] + 1);

    point[0] = -0.6;
    point[2] = 0.75;
    CopyVector(point, (*output)[2] + 2);

    point[0] = -0.1;
    CopyVector(point, (*output)[2] + 3);

    point[2] = 0.8;
    CopyVector(point, (*output)[2] + 4);

    point[0] = -0.53;
    CopyVector(point, (*output)[2] + 5);

    point[0] = -0.65;
    point[2] = 1.0;
    CopyVector(point, (*output)[2] + 6);

    point[0] = -0.1;
    CopyVector(point, (*output)[2] + 7);
```

```
    point[2] = 1.05;
    CopyVector(point, (*output)[2] + 8);

    point[0] = -1.1;
    CopyVector(point, (*output)[2] + 9);
}

//duplicate 3 previous shapes, rotating by pi around z
{
    matrix4 rotationMatrix;
    DefineElementaryTransform(&rotationMatrix, ROTATE_Z, PI);
    for(i = 0; i < 3; i++) {
        for(j = 0; j < (*sizes)[i]; j++) {
            ApplyTransform((*output)[i][j], rotationMatrix, ((*output)[i + 3] + j));
        }
    }
}

//draw long rectangle across center
{
    point[0] = -1.0;
    point[2] = 0.55;
    CopyVector(point, (*output)[6] + 0);

    point[0] = 1.0;
    CopyVector(point, (*output)[6] + 1);

    point[2] = 0.65;
    CopyVector(point, (*output)[6] + 2);

    point[0] = -1.0;
    CopyVector(point, (*output)[6] + 3);
}

//draw big trapezoid up top
{
    point[0] = -0.65;
    point[2] = 1.1;
    CopyVector(point, (*output)[7] + 0);

    point[0] = 0.65;
    CopyVector(point, (*output)[7] + 1);

    point[0] = 0.2;
```

```
        point[2] = 1.2;
        CopyVector(point, (*output)[7] + 2);

        point[0] = -0.2;
        CopyVector(point, (*output)[7] + 3);
    }
}

//Generates shapes representing the word Spencer
//Returns total of 7 shapes
//Number of points in each shape is stored in sizes
void DataSpencer(vector4 *output[7], int sizes[7]) {
    int i;
    vector4 point;
    matrix4 translateMatrix;

    point[0] = 0;
    point[1] = 0;
    point[2] = 0;
    point[3] = 1;

    sizes[0] = 12;
    sizes[1] = 10;
    sizes[2] = 12;
    sizes[3] = 8;
    sizes[4] = 8;
    sizes[5] = sizes[2];
    sizes[6] = 8;
    for(i = 0; i < 7; i++) {
        output[i] = (vector4 *) malloc(sizeof(vector4) * sizes[i]);
    }

    DefineElementaryTransform(&translateMatrix, IDENTITY, 0);
    //s
    {
        vector4 points[12] = {
            {0, 0},
            {0.8, 0},
            {0.8, 0.55},
            {0.1, 0.55},
            {0.1, 0.9},
            {0.8, 0.9},
            {0.8, 1.0},
            {0, 1.0},
            {0, 0.45},
```

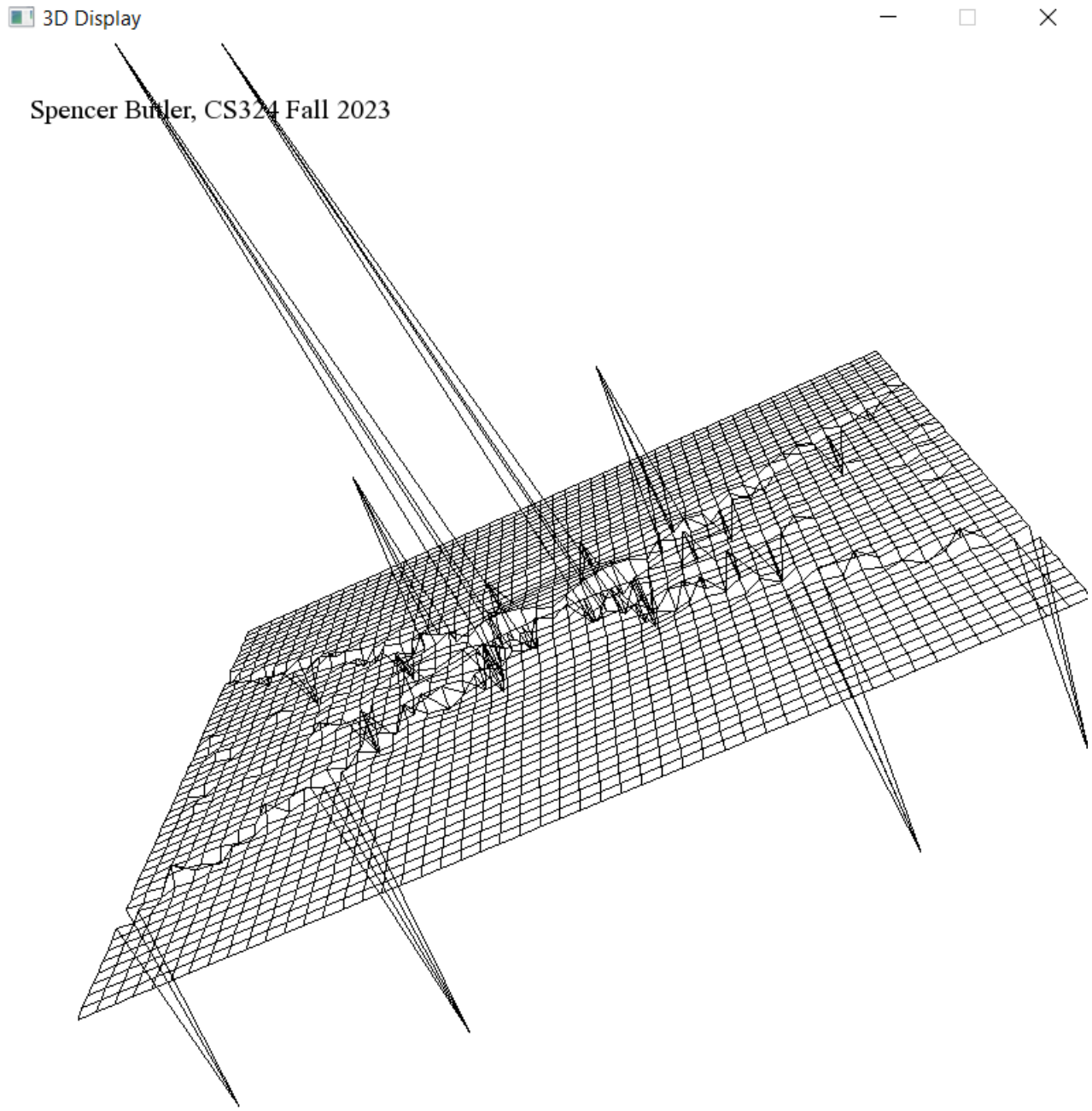
```
        {0.7, 0.45},
        {0.7, 0.1},
        {0, 0.1}
    };
    for(i = 0; i < sizes[0]; i++) {
        point[0] = points[i][0];
        point[2] = points[i][1];
        ApplyTransform(point, translateMatrix, output[0] + i);
    }
}

//p
{
    BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
    vector4 points[10] = {
        {0, -1},
        {0.1, -1},
        {0.1, 0.9},
        {0.7, 0.9},
        {0.7, 0.1},
        {0.1, 0.1},
        {0.1, 0},
        {0.8, 0},
        {0.8, 1},
        {0, 1}
    };
    for(i = 0; i < sizes[1]; i++) {
        point[0] = points[i][0];
        point[2] = points[i][1];
        ApplyTransform(point, translateMatrix, output[1] + i);
    }
}

//e
{
    BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
    vector4 points[12] = {
        {0.1, 0.55},
        {0.7, 0.55},
        {0.7, 0.9},
        {0.1, 0.9},
        {0.1, 0.1},
        {0.8, 0.1},
        {0.8, 0},
        {0, 0},
```

```
        {0, 1.0},
        {0.8, 1.0},
        {0.8, 0.45},
        {0.1, 0.45}
    };
    for(i = 0; i < sizes[2]; i++) {
        point[0] = points[i][0];
        point[2] = points[i][1];
        ApplyTransform(point, translateMatrix, output[2] + i);
    }
}

//n
{
    BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
    vector4 points[8] = {
        {0, 0},
        {0, 1},
        {0.8, 1},
        {0.8, 0},
        {0.7, 0},
        {0.7, 0.9},
        {0.1, 0.9},
        {0.1, 0}
    };
    for(i = 0; i < sizes[3]; i++) {
        point[0] = points[i][0];
        point[2] = points[i][1];
        ApplyTransform(point, translateMatrix, output[3] + i);
    }
}

//c
{
    BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
    vector4 points[8] = {
        {0.8, 0},
        {0, 0},
        {0, 1},
        {0.8, 1},
        {0.8, 0.9},
        {0.1, 0.9},
        {0.1, 0.1},
        {0.8, 0.1}
    };
```

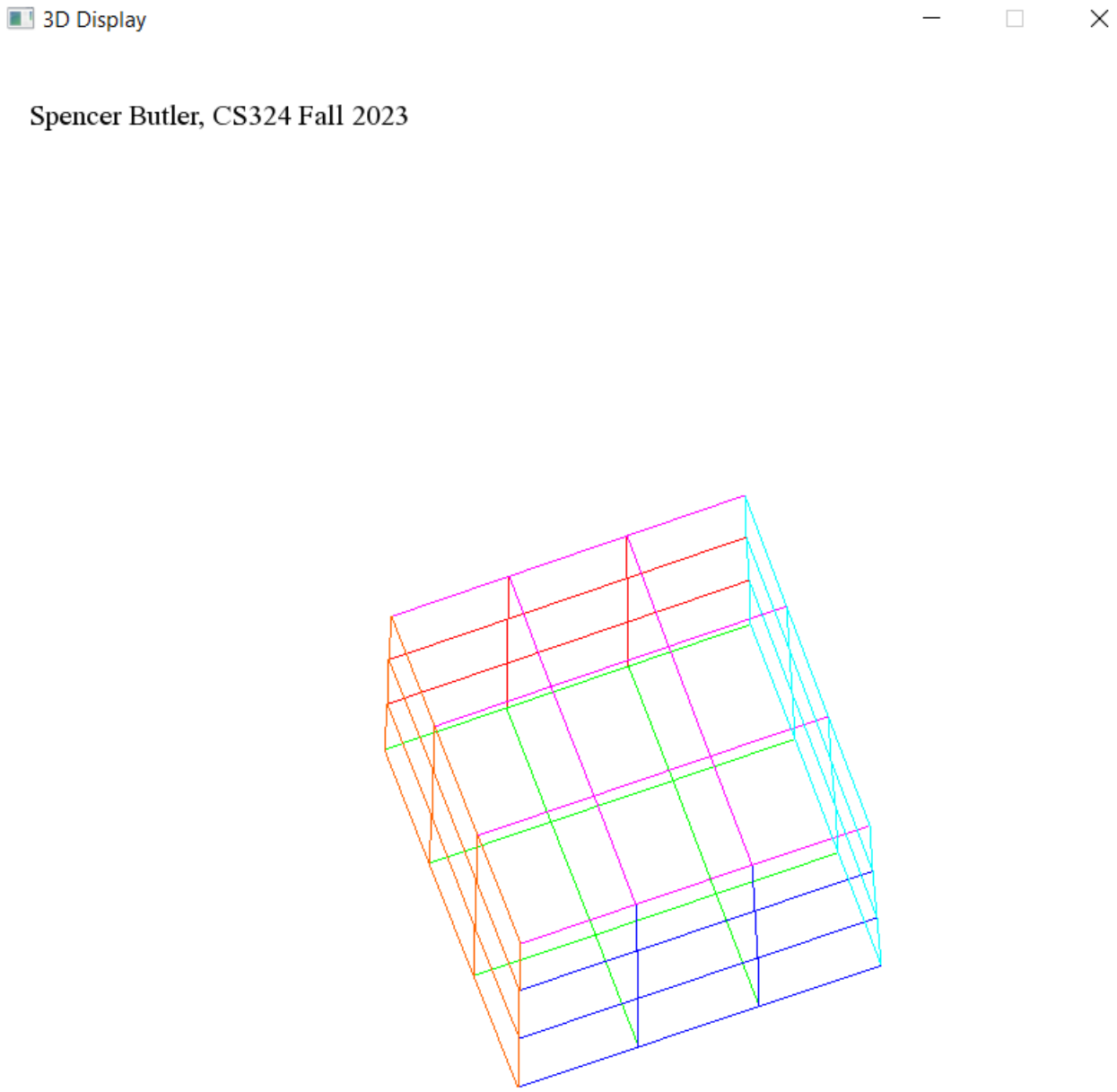
```
        for(i = 0; i < sizes[4]; i++) {
            point[0] = points[i][0];
            point[2] = points[i][1];
            ApplyTransform(point, translateMatrix, output[4] + i);
        }
    }

    //copy e from output[2]
    {
        BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
        matrix4 offsetMatrix;
        DefineElementaryTransform(&offsetMatrix, TRANSLATE_X, 3);
        for(i = 0; i < sizes[5]; i++) {
            ApplyTransform(output[2][i], offsetMatrix, output[5] + i);
        }
    }

    //r
    {
        BuildElementaryTransform(&translateMatrix, TRANSLATE_X, 1);
        vector4 points[8] = {
            {0, 0},
            {0, 1},
            {0.8, 1},
            {0.8, 0.7},
            {0.7, 0.7},
            {0.7, 0.9},
            {0.1, 0.9},
            {0.1, 0}
        };
        for(i = 0; i < sizes[6]; i++) {
            point[0] = points[i][0];
            point[2] = points[i][1];
            ApplyTransform(point, translateMatrix, output[6] + i);
        }
    }
}
```

4 Output

4.1 The function:

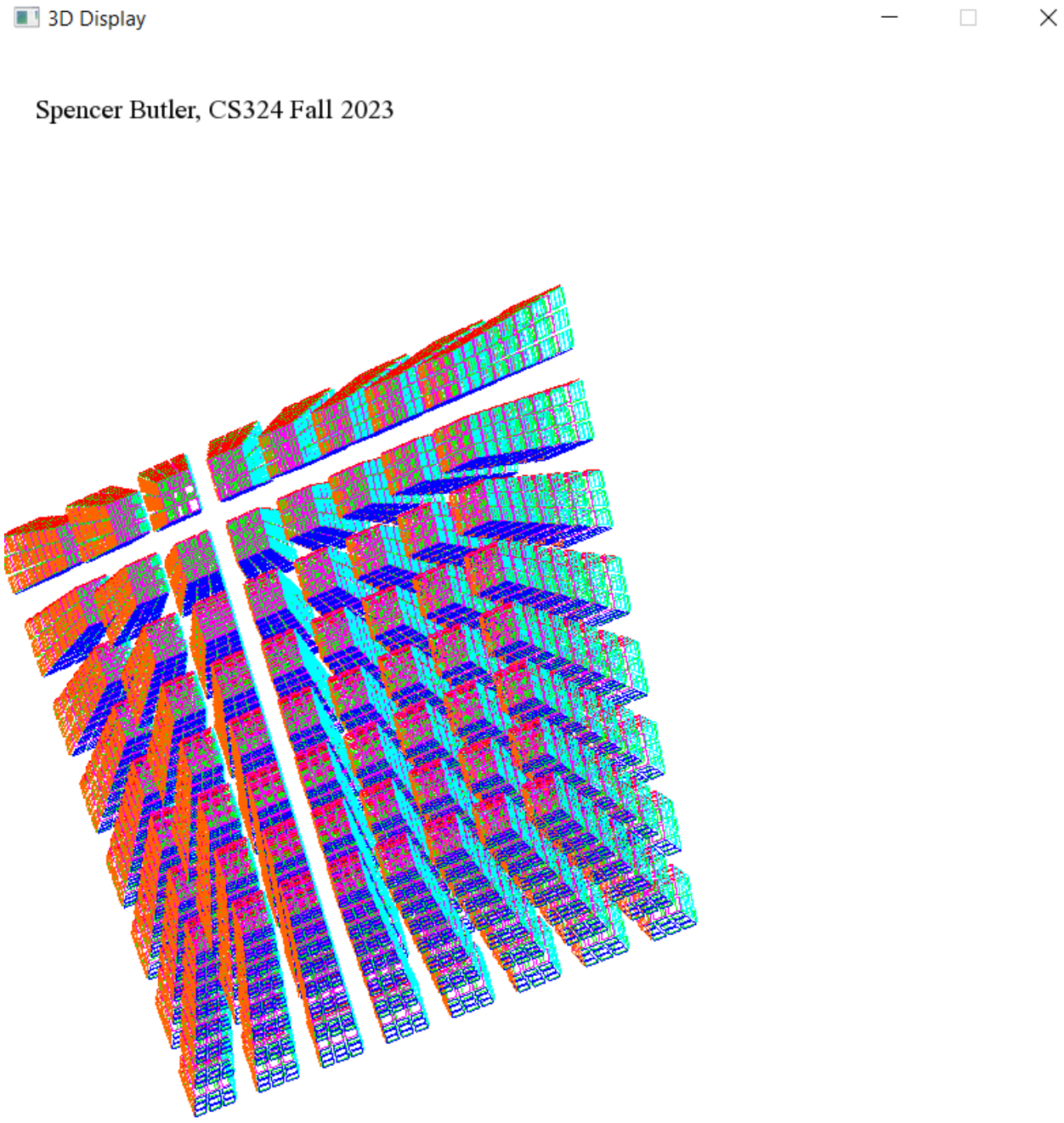


4.2 Rubik's cube with no gaps:

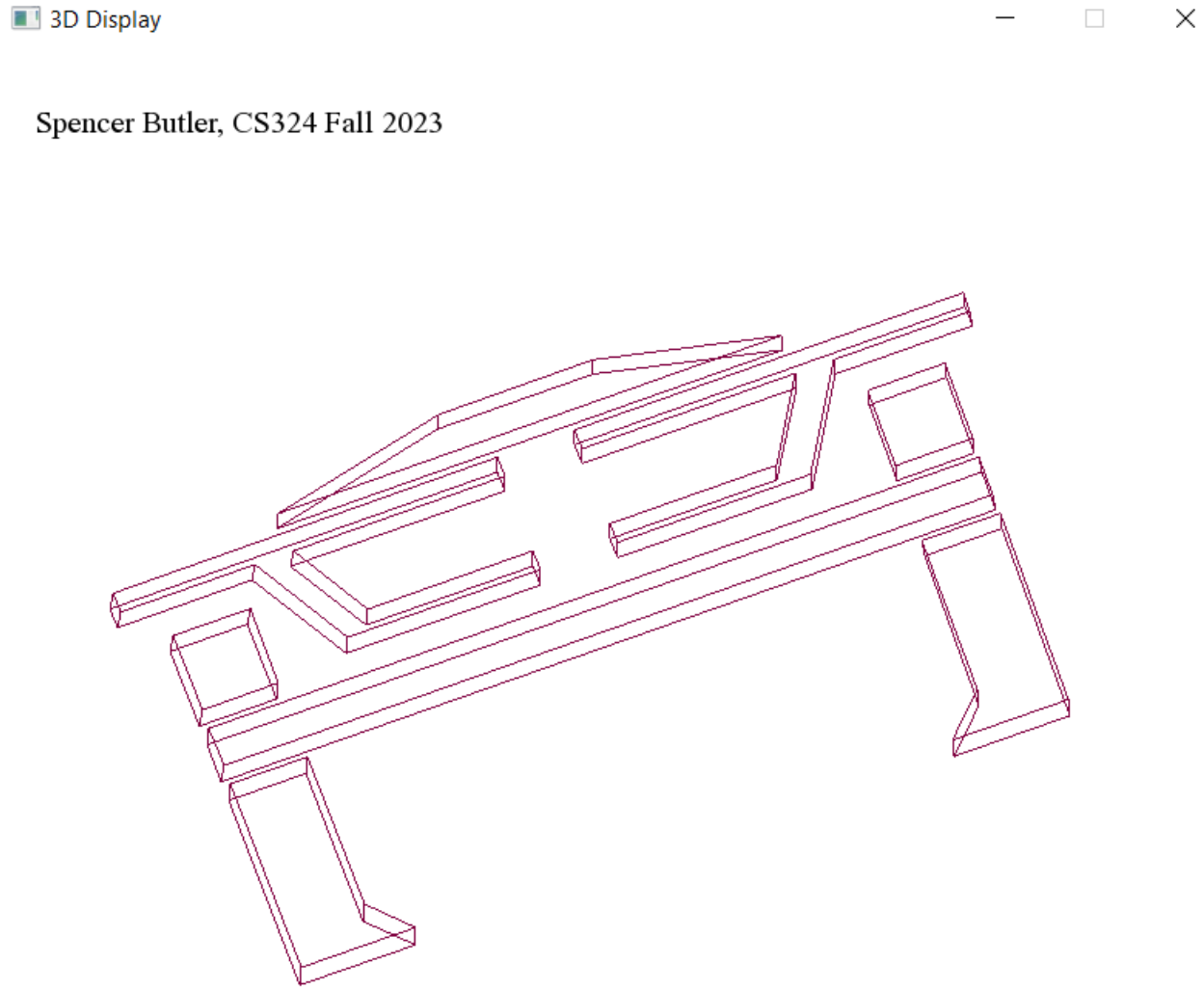


4.3 Rubik's cube with gaps:

4.4 Multiple Rubik's cubes with gaps:



4.5 Tron recognizer:



4.6 My name in block letters: