SCHWEIZER JUGEND FORSCHT
LA SCIENCE APPELLE LES JEUNES
SCIENZA E GIOVENTÙ
SCIENZA E GIUVENTETGNA

Università della Svizzera italiana

Faculty of Informatics

TETRIS 3D GAME DESIGN

a project of: Simon Erni & Robin Vaaler
assisted by: Luca Colombo & Randolf Schärfig

The project

With the help of the tutors, we managed to create a 3D Tetris with all necessary functions for a great retro-game experience.

3D Matrix

We used a matrix to store the data of the current position and colour of the pieces and its blocks. This concept allows us to easily modify the position of the pieces and provides enough speed for all the testing and rendering algorithms (checking if the move is allowed, see if there is a block and if yes, what colour does it have.)
Actually, we used 2 different matrices. One for storing the blocks already fallen down, the other holds the data of the piece currently falling down.

Collision detection algorithm

When the pieces are falling down, a collision detection algorithm checks if there is already a block underneath, so moving one down would lead to overlapping blocks. Then, it would simply redo the move. Also, when the pieces are moved or rotated, it checks if at the desired new position there is already a block and in case, it simply not allow the move or rotation.

Rotation algorithm

For rotating a piece stored in a matrix, you can either rotate the whole matrix or just switch the axis: we opted for the second option.
When a piece is rotated, the matrix remains completely untouched. The rendering function accessing this matrix simply changes the way of accessing it.
When you want to rotate a whole matrix, an easy way of doing it would be to simply exchange the x and y axis (just considering 2D rotation) and depending on the side of the rotation, making one axis minus. Thus, only the actual rotation state is stored and modified when the player chooses to rotate the piece. You can apply this also when third dimension (z axis) comes into play.

Moving and Gravity

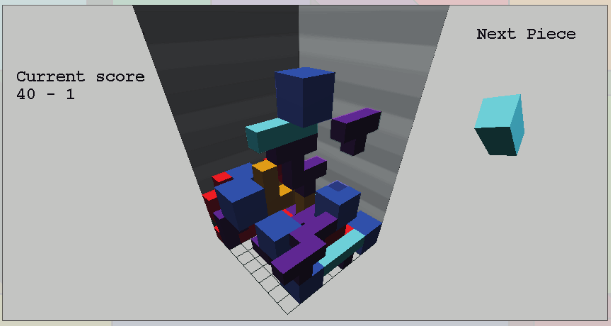
Moving a piece in the game field is quite easy. You only have to adjust the position of the overlying matrix, the one which stores the piece currently falling down. Gravity is nothing else than moving the piece after a specific amount of time one step down.

Timer

We used a timer to control the gravity, which depends on the current level of the player. If the level of the player increases, then the piece will fall down faster. You just have to adjust the interval of the timer when the player goes one level up.

Camera

The camera movement depends on the mouse movement of the user. The difference of the position of the mouse is measured and then applied into a camera movement around the centre of the game field. This causes the "illusion" of hovering around the game field.



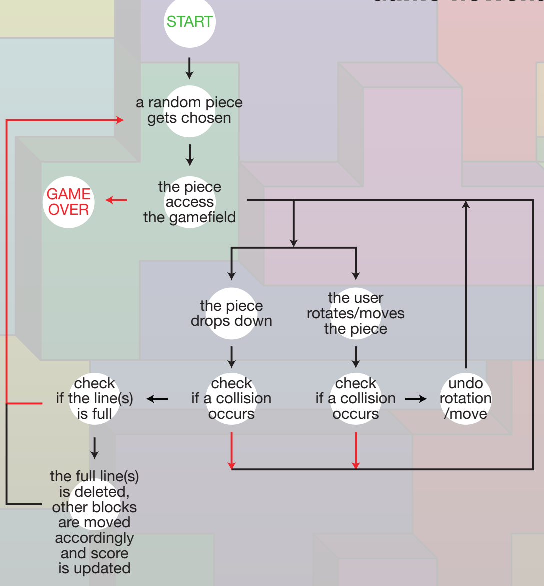
Programming language: C++

For developing this 3D Tetris we used the programming language C++, with the QT libraries for displaying the window and handling user inputs and OpenGL for the 3D rendering. Our tutors created the framework of the program, then we implemented almost all the functions of the program.
What we appreciated of C++ is that is an Object Oriented programming language: and it was the first time we programmed using classes
C++ provides classes and instances for storing functions (methods) and variables (properties), and this is very helpful for cleaning and organizing the source code when you deal with a complex project. With C++, you can also import libraries, and use them to simplify the development of projects like our (we used OpenGL and QT libraries). On top of that, they're Open Source hence completely free to use.

A bit of source code

```
void BasicPiece::applyRotation(char a, int s) {
    Vector3D cmin, cmax;
    cmin = min;
    cmax = max;
    int cache;
    // update the min,max accordingly!
    if (a == 'z'){
        cache = xAxis;
        if (s<0){ xAxis = yAxis; yAxis = -cache; min.y = 4 - cmax.x; max.y = 4 -cmin.x; min.x = cmin.y; max.x = cmax.y;}
        else {xAxis = -yAxis; yAxis = cache; min.y = cmin.x; max.y = cmax.x; min.x = 4 - cmax.y; max.x = 4 - cmin.y;}
    }
    if (a == 'y'){
        cache = xAxis;
        if (s<0){ xAxis = zAxis; zAxis = -cache; min.z = 4 - cmax.x; max.z = 4 -cmin.x; min.x = cmin.z; max.x = cmax.z;}
        else {xAxis = -zAxis; zAxis = cache; min.z = cmin.x; max.z = cmax.x; min.x = 4 - cmax.z; max.x = 4 - cmin.z;}
    }
    if (a == 'x'){
        cache = yAxis;
        if (s<0){ yAxis = zAxis; zAxis = -cache; min.z = 4 - cmax.y; max.z = 4 -cmin.y; min.y = cmin.z; max.y = cmax.z;}
        else {yAxis = -zAxis; zAxis = cache; min.z = cmin.y; max.z = cmax.y; min.y = 4 - cmax.z; max.y = 4 - cmin.z;}
    }
}

bool BasicPiece::checkMove() {
    Vector3D tmax, tmin;
    tmax = trans + max;
    tmin = trans + min;

    if (tmax.y > FIELD_HEIGHT || tmax.x >= FIELD_WIDTH || tmax.z >= FIELD_DEPTH) {return false;}
    if (tmin.y < 0 || tmin.x < 0 || tmin.z < 0) {return false;}

    for (int x = min.x; x <= max.x; x++) {
        for (int y = min.y ; y <= max.y; y++) {
            for (int z = min.z; z <= max.z; z++) {
                RotCoordSys cord;
                cord = getRot(x,y,z);
                if ((*field)[x + trans.x][y + trans.y][z + trans.z].isOccupied && P.mat[cord.x][cord.y][cord.z]) {
                    return false;
                }
            }
        }
    }
    return true;
}

bool BasicPiece::rotate(char a, int s) {
    // apply rotation and uses the function checkmove that checks for collision or out of bounds
    applyRotation(a,s);
    if (!checkMove()) { applyRotation(a,-s); return false;}
    return true;
}
```

Game flowchart


START
a random piece gets chosen
the piece access the gamefield
GAME OVER
the piece drops down
the user rotates/moves the piece
check if a collision occurs
check if a collision occurs
undo rotation /move
check if the line(s) is full
the full line(s) is deleted, other blocks are moved accordingly and score is updated