Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχ. Η/Υ και Πληροφορικής Χειμερινό Εξάμηνο 2024

ΜΥΥ702

Γραφικά Υπολογιστών & Συστήματα Αλληλεπίδρασης

Διδάσκων: Ιωάννης Φούντος

Υπεύθυνη εργαστηριακού μέρους μαθήματος: Βασιλική Σταμάτη

Προγραμματιστική Άσκηση 1-Α

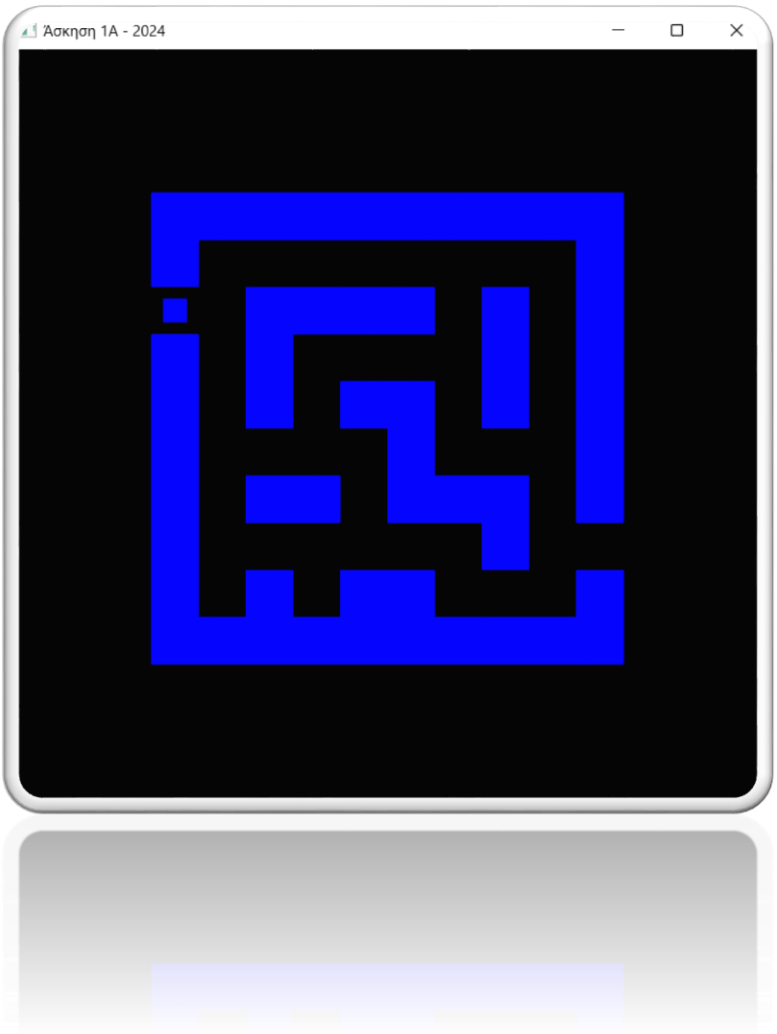
OpenGL 2024-2025

Αναφορά

Παναγιώτης Παρασκευόπουλος

ΑΜ:2905

Περιεχόμενα



Περιγραφή της εργασίας

Η συγκεκριμένη αναφορά βασίζεται στην πρώτη προγραμματιστική άσκηση OpenGL. Σκοπός αυτής της άσκησης είναι να εξοικειωθούμε με την χρήση βασικών βιβλιοθηκών στοιχειωδών γραφικών της Opengl οι οποίες υποστηρίζουν 2Δ και 3Δ γραφικά (μόνο τις βιβλιοθήκες GLEW, GLFW και GLM). Στην άσκηση αυτή θα δημιουργήσουμε ένα παράθυρο στο οποίο θα ζωγραφίζουμε ένα λαβύρινθο και στη συνέχεια έναν χαρακτήρα Α, ο οποίος θα κινείται μέσα στον λαβύρινθο. Η κίνησή του θα ελέγχεται από το πληκτρολόγιο του χρήστη.

Η άσκηση αυτή έγινε από ένα άτομο. Στην αναφορά χρησιμοποιώ α’ πληθυντικό για καλύτερη ανάγνωση.

Ερώτημα (i)

Φτιάχνουμε το πρόγραμμα μας να ανοίγει ένα βασικό παράθυρο με μέγεθος 750 x 750 pixels και να έχει τίτλο “Άσκηση 1Α - 2024”.

```
149     // Open a window and create its OpenGL context
150     window = glfwCreateWindow(750, 750, u8"Άσκηση 1Α - 2024", NULL, NULL);
```

Το background του παραθύρου πρέπει να είναι μαύρο, οπότε αλλάζουμε όλες τις τιμές σε 0.0.

```
173     // Black background
174     glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

Βάζουμε την εφαρμογή μας να τερματίζει οποιαδήποτε στιγμή πατώντας το πλήκτρο Q.

```
392     while (glfwGetKey(window, GLFW_KEY_Q) != GLFW_PRESS &&  glfwWindowShouldClose(window) == 0);
```

Ερώτημα (ii)

Το πρόγραμμα ξεκινάει ζωγραφίζοντας έναν λαβύρινθο. Ο λαβύρινθος σχηματίζεται ζωγραφίζοντας τετράγωνα μπλε χρώματος, που αντιστοιχούν στα τοιχώματα του λαβύρινθου. Οπότε αρχικά πηγαίνουμε στο αρχείο ProjectFragmentShader και αλλάζουμε τις τιμές σε (0,0,1) για να έχουμε την απόχρωση του μπλε.

```
// Output color = blue
color = vec3(0,0,1);
```

Στο αρχείο Source-1A.cpp, μέσα στη main, ορίζουμε τον maze ως έναν 2Δ πίνακα που αναπαριστά το λαβύρινθο ως πλέγμα 10x10 που περιέχει τιμές 0 ή 1, όπως στην εικόνα 2 της εκφώνησης. Η τιμή “1” αντιπροσωπεύει τοίχους και η τιμή “0” κενά (μονοπάτι).

```
207     // Ορισμός του λαβυρίνθου (10x10 grid)
208     const int maze[10][10] = {
209         {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
210         {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
211         {0, 0, 1, 1, 1, 1, 0, 1, 0, 1},
212         {1, 0, 1, 0, 0, 0, 0, 1, 0, 1},
213         {1, 0, 1, 0, 1, 1, 0, 1, 0, 1},
214         {1, 0, 0, 0, 0, 1, 0, 0, 0, 1},
215         {1, 0, 1, 1, 0, 1, 1, 1, 0, 1},
216         {1, 0, 0, 0, 0, 0, 0, 1, 0, 0},
217         {1, 0, 1, 0, 1, 1, 0, 0, 0, 1},
218         {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
219     };
```

Με το `std::vector<GLfloat> maze_vertices;` αποθηκεύουμε τις κορυφές (vertices) των τοίχων του λαβύρινθου, που θα χρησιμοποιηθούν για την σχεδίαση τους.

Ορίζουμε το μέγεθος κάθε τετραγώνου να είναι 1.0.

```
// Μέγεθος κάθε τετραγώνου
float size = 1.0;
```

Στη συνέχεια με ένα loop διατρέχουμε τα στοιχεία του πίνακα maze, δηλαδή τις 10 γραμμές και 10 στήλες.

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
```

Ελέγχουμε αν το στοιχείο είναι “1” (δηλαδή τοίχος).

```
if (maze[i][j] == 1) {
```

Αν είναι, προχωράμε στην δημιουργία των κορυφών. Για κάθε τοίχο του λαβύρινθου δημιουργούμε 2 τρίγωνα για να σχηματίσουν τετράγωνο.

```
maze_vertices.push_back(x);          maze_vertices.push_back(y);          maze_vertices.push_back(0.0f);
maze_vertices.push_back(x + size);   maze_vertices.push_back(y);          maze_vertices.push_back(0.0f);
maze_vertices.push_back(x + size);   maze_vertices.push_back(y + size);   maze_vertices.push_back(0.0f);

maze_vertices.push_back(x);          maze_vertices.push_back(y + size);   maze_vertices.push_back(0.0f);
maze_vertices.push_back(x);          maze_vertices.push_back(y);          maze_vertices.push_back(0.0f);
maze_vertices.push_back(x + size);   maze_vertices.push_back(y + size);   maze_vertices.push_back(0.0f);
```

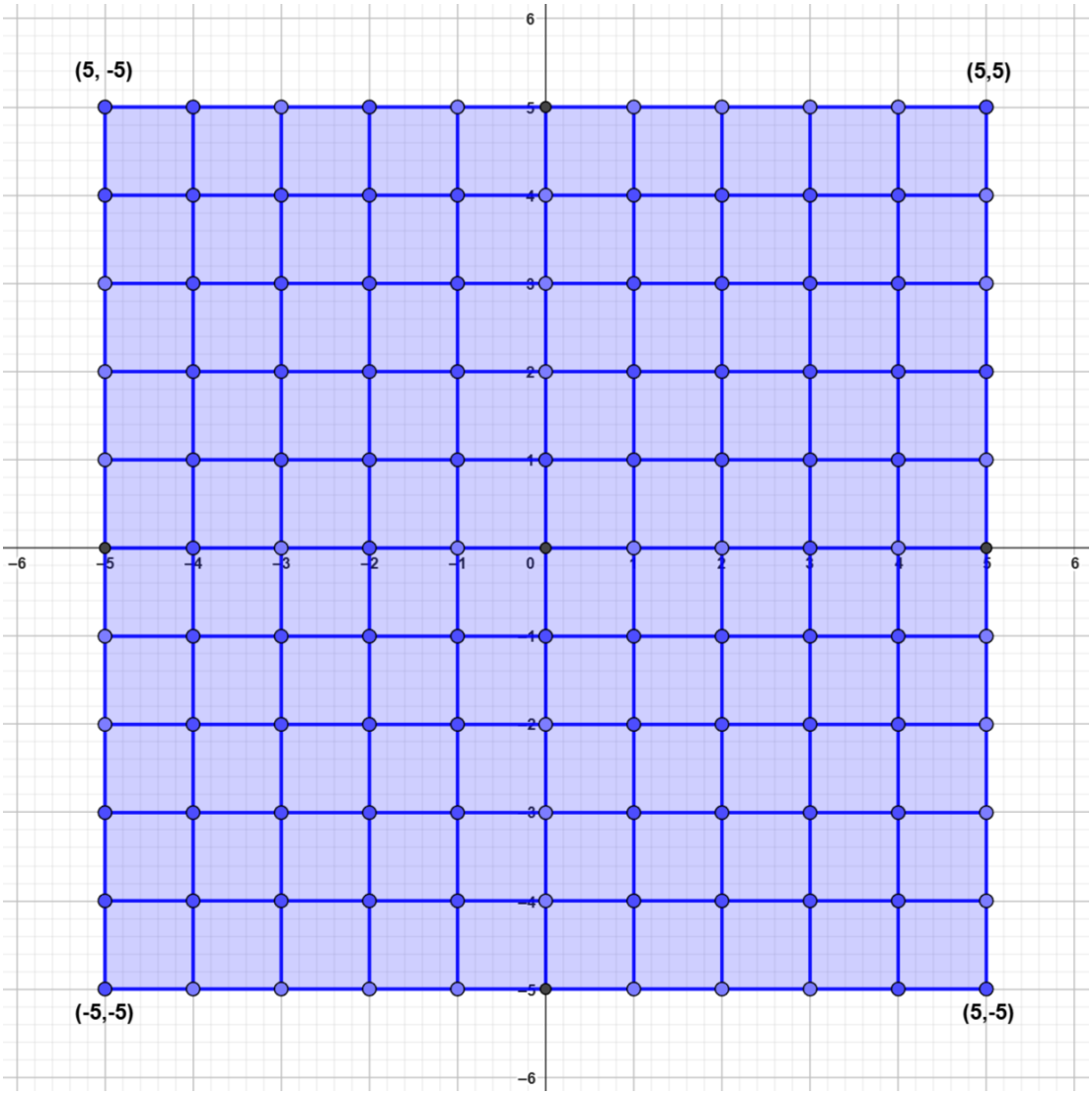
Κάθε τοίχος (1) αντιστοιχεί σε 6 κορυφές (2 τρίγωνα) που προστίθενται στο maze_vertices. Το πρώτο τρίγωνο σχηματίζεται από τις τρεις πρώτες κορυφές:
Κορυφή 1: (x, y) (κάτω αριστερά)
Κορυφή 2: (x + size, y) (κάτω δεξιά)
Κορυφή 3: (x + size, y + size) (πάνω δεξιά)

Το δεύτερο τρίγωνο σχηματίζεται από τις τρεις επόμενες κορυφές:
Κορυφή 4: (x , y + size) (πάνω αριστερά)
Κορυφή 5: (x, y) (κάτω αριστερά)
Κορυφή 6: (x + size, y + size) (πάνω δεξιά)

Μετατοπίζουμε τις συντεταγμένες έτσι ώστε ο λαβύρινθος να κεντραριστεί γύρω από το σημείο (0,0).

```
float x = (j - 5) * size; // Με
float y = ((9 - i) - 5) * size;
```

Με το (j – 5) μετατοπίζουμε τον τοίχο κατά 5 μονάδες προς τα αριστερά ή δεξιά. Με το (i – 5) μετατοπίζουμε τον τοίχο κατά 5 μονάδες προς τα πάνω ή κάτω. Επειδή ο λαβύρινθος εμφανιζόταν ανεστραμμένος , χρειάστηκε να βάλουμε (9 – i) για να αντιστρέψουμε την κατεύθυνση του άξονα y, καθώς οι γραμμές του πίνακα διατρέχουν από πάνω προς τα κάτω.



Δημιουργούμε και τα buffers OpenGL για την αποθήκευση των γεωμετρικών δεδομένων του λαβυρίνθου.

```
247     GLuint vertexbuffer;
248     glGenBuffers(1, &vertexbuffer);
249     glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
250     glBufferData(GL_ARRAY_BUFFER, maze_vertices.size() * sizeof(GLfloat), maze_vertices.data(), GL_STATIC_DRAW);
```

Μέσα στη do φτιάχνουμε την συνάρτηση και ενημερώνουμε και την glDrawArrays για τον σχεδιασμό των τριγώνων του λαβύρινθου στην οθόνη.

```
359             glEnableVertexAttribArray(0);
360             glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
361             glVertexAttribPointer(
362                 0,                  // attribute 0, must match the layout in the shader.
363                 3,                  // size
364                 GL_FLOAT,           // type
365                 GL_FALSE,           // normalized?
366                 0,                  // stride
367                 (void*)0            // array buffer offset
368             );
369             glDrawArrays(GL_TRIANGLES, 0, maze_vertices.size() / 3);
370             glDisableVertexAttribArray(0);
```

Ερώτημα (iii)

Θέλουμε να δημιουργήσουμε ένα μικρότερο μπλε τετράγωνο (εμείς εδώ το λέμε παίκτη), αντιστοιχεί σε έναν κινούμενο αντικείμενο που θα διασχίζει τον λαβύρινθο.
Αρχικά ορίζουμε την θέση του παίκτη στον λαβύρινθο.

```
254     int player_x = 0; // στήλη 0
255     int player_y = 2; // γραμμή 2
```

Οι μεταβλητές player_x και player_y αρχικοποιούνται με τιμές 0 και 2, αντίστοιχα, γιατί ο παίκτης θέλουμε να ξεκινάει από το πρώτο κελί της τρίτης γραμμής του λαβύρινθου.

Φτιάχνουμε τη συνάρτηση update_square_A_vertices, η οποία υπολογίζει τις κορυφές ενός τετραγώνου βασισμένου στη θέση του παίκτη.

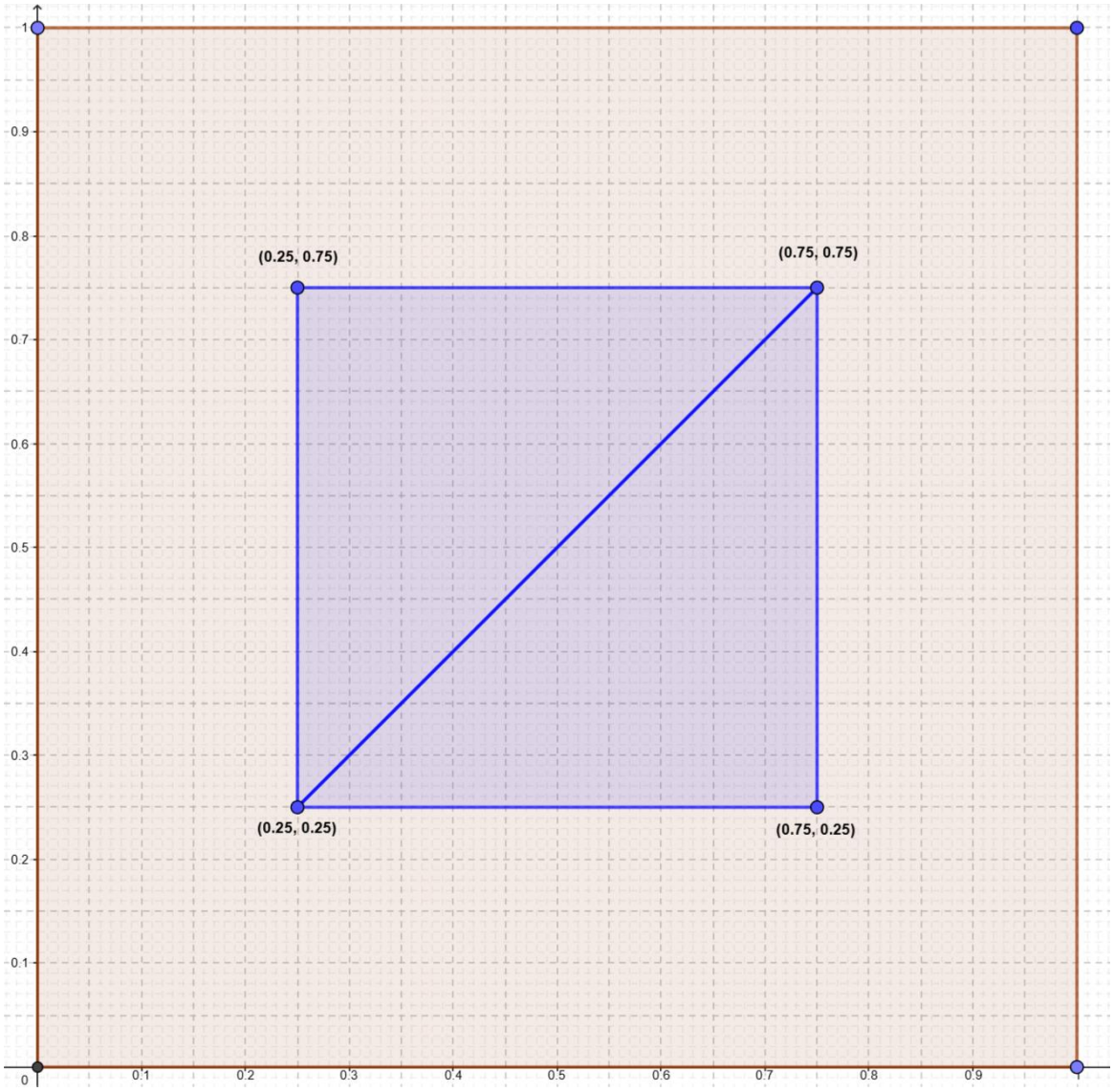
```
258     auto update_square_A_vertices = [&](int x, int y) -> std::vector<GLfloat> {
259         float cell_size = 1.0f;
260         float center_x = (x - 5) * cell_size;
261         float center_y = ((9 - y) - 5) * cell_size;
```

Ορίζουμε το cell_size ως 1.0f και αντιπροσωπεύει το μέγεθος κάθε κελιού στο grid.
Ορίζουμε τα center_x και center_y τα οποία υπολογίζουν τη θέση του κέντρου του κελιού με βάση τις συντεταγμένες (x, y).

Η συνάρτηση επιστρέφει έναν std::vector<GLfloat>, ο οποίος περιέχει τις τρισδιάστατες συντεταγμένες για τις 6 κορυφές του τετραγώνου.
Αυτές οι κορυφές σχεδιάζουν ένα τετράγωνο χρησιμοποιώντας δύο τρίγωνα.
Πρώτο τρίγωνο: κάτω αριστερή, κάτω δεξιά, πάνω δεξιά κορυφή.
Δεύτερο τρίγωνο: πάνω αριστερή, πάνω δεξιά, κάτω αριστερή κορυφή.

```
263     return {
264         center_x + 0.25f, center_y + 0.25f, 0.0f,  // κάτω αριστερή κορυφή
265         center_x + 0.75f, center_y + 0.25f, 0.0f,  // κάτω δεξιά κορυφή
266         center_x + 0.75f, center_y + 0.75f, 0.0f,  // πάνω δεξιά κορυφή
267
268         center_x + 0.25f, center_y + 0.75f, 0.0f,  // πάνω αριστερή κορυφή
269         center_x + 0.75f, center_y + 0.75f, 0.0f,  // πάνω δεξιά κορυφή
270         center_x + 0.25f, center_y + 0.25f, 0.0f   // κάτω αριστερή κορυφή
271     };
```

Χρειάστηκε να βάλουμε τις κατάλληλες συντεταγμένες κάθε κορυφής για να κεντραριστεί το τετράγωνο με πλευρά 0.5 στο κέντρο του κελιού.



Η square_A_vertices παίρνει την τιμή που επιστρέφει η update_square_A_vertices (player_x, player_y), δηλαδή τις αρχικές κορυφές του τετραγώνου που αντιστοιχούν στη θέση του παίκτη.

```
275        std::vector<GLfloat> square_A_vertices = update_square_A_vertices(player_x, player_y);
```

Δημιουργούμε και τα buffers OpenGL για την αποθήκευση των γεωμετρικών δεδομένων του τετραγώνου.

```
277        GLuint square_vertexbuffer;
278        glGenBuffers(1, &square_vertexbuffer);
279        glBindBuffer(GL_ARRAY_BUFFER, square_vertexbuffer);
280        glBufferData(GL_ARRAY_BUFFER, square_A_vertices.size() * sizeof(GLfloat), square_A_vertices.data(), GL_STATIC_DRAW);
```

Όπως και στην περίπτωση του λαβύρινθου, μέσα στη do φτιάχνουμε την συνάρτηση και ενημερώνουμε και την glDrawArrays για τον σχεδιασμό των τριγώνων του τετραγώνου στην οθόνη.

```
373         glBindBuffer(GL_ARRAY_BUFFER, square_vertexbuffer);
374         glVertexAttribPointer(
375             0,                  // attribute 0, must match the layout in the shader.
376             3,                  // size
377             GL_FLOAT,           // type
378             GL_FALSE,           // normalized?
379             0,                  // stride
380             (void*)0            // array buffer offset
381         );
382         glEnableVertexAttribArray(0);
383         glDrawArrays(GL_TRIANGLES, 0, 6);  // 6 κορυφές για το τετράγωνο (2 τρίγωνα)
384         glDisableVertexAttribArray(0);
```

Επίσης, ενημερώνουμε τις κορυφές του τετραγώνου και τον buffer με τις νέες κορυφές, όταν αυτό αλλάξει θέση (κελί).

```
351         // Ενημέρωση κορυφών του τετραγώνου A
352         square_A_vertices = update_square_A_vertices(player_x, player_y);
353
354         // Ενημέρωση του buffer με τις νέες κορυφές
355         glBindBuffer(GL_ARRAY_BUFFER, square_vertexbuffer);
356         glBufferData(GL_ARRAY_BUFFER, square_A_vertices.size() * sizeof(GLfloat), square_A_vertices.data(), GL_STATIC_DRAW);
```

Ερώτημα (iv)

Τώρα θέλουμε ο χαρακτήρας A να κινείται μέσα στον λαβύρινθο. Η κίνησή του ελέγχεται από το πληκτρολόγιο του χρήστη, και συγκεκριμένα:
Αν πατηθεί το πλήκτρο L, κινείται μία θέση δεξιά.
Αν πατηθεί το πλήκτρο J, κινείται μία θέση αριστερά.
Αν πατηθεί το πλήκτρο K, κινείται μία θέση προς τα κάτω.
Αν πατηθεί το πλήκτρο I, κινείται μία θέση προς τα πάνω.

Ορίζουμε έξω από την do τις λογικές μεταβλητές που αποθηκεύουν την κατάσταση του κάθε πλήκτρου. Αν μια μεταβλητή είναι true, σημαίνει ότι το αντίστοιχο πλήκτρο είναι πατημένο. Αυτή η κατάσταση αποτρέπει την επανειλημμένη κίνηση σε κάθε ανανέωση του προγράμματος όσο το πλήκτρο παραμένει πατημένο.

```
282         bool key_l_pressed = false;
283         bool key_j_pressed = false;
284         bool key_k_pressed = false;
285         bool key_i_pressed = false;
```

Δημιουργούμε μέσα στην do οκτώ βασικές if (δύο για κάθε πλήκτρο), και με την συνάρτηση glfwGetKey (όπως είχαμε χρησιμοποιήσει για να τερματίσουμε το πρόγραμμα με το πλήκτρο Q) ελέγχουμε αν το πλήκτρο είναι πατημένο ή όχι.

Αναλύουμε για το πλήκτρο “L”

```
301         if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS && !key_l_pressed) {
302             if (player_x + 1 < 10 && maze[player_y][player_x + 1] == 0) { // κίνηση δεξιά
303                 player_x++;
304             }
305             key_l_pressed = true; // Σημείωσε ότι το πλήκτρο είναι τώρα πατημένο
306         }
307         if (glfwGetKey(window, GLFW_KEY_L) == GLFW_RELEASE) {
308             key_l_pressed = false; // Επαναφορά όταν το πλήκτρο απελευθερωθεί
309         }
```

Στο πρώτο if ελέγχουμε αν το πλήκτρο L είναι πατημένο και η μεταβλητή key_l_pressed είναι false (δηλαδή, δεν είχε προηγουμένως καταγραφεί ότι το πλήκτρο είναι πατημένο), τότε:
Ελέγχουμε αν ο παίκτης μπορεί να κινηθεί δεξιά χωρίς να βγει από τα όρια του λαβύρινθου (player_x + 1 < 10), και αν η θέση προς τα δεξιά είναι κενή (maze[player_y][player_x + 1] == 0). Αν οι συνθήκες ισχύουν, αυξάνει τη συντεταγμένη x του παίκτη (player_x++), μετακινώντας τον μία θέση δεξιά. Μετά την κίνηση, ορίζουμε την μεταβλητή key_l_pressed ως true για να σημειώσει ότι το πλήκτρο έχει καταγραφεί ως πατημένο, αποτρέποντας την επανειλημμένη κίνηση μέχρι να απελευθερωθεί.
Στο δεύτερο if ελέγχουμε αν το πλήκτρο L έχει απελευθερωθεί (GLFW_RELEASE) και τότε η key_l_pressed γίνεται false, επιτρέποντας έτσι μια νέα κίνηση με επόμενο πάτημα.

Το αντίστοιχο γίνεται και για τα υπόλοιπα πλήκτρα με τις διαφορές ότι:

Για το πλήκτρο J, ελέγχουμε αν ο παίκτης μπορεί να κινηθεί αριστερά. Αν μπορεί μειώνουμε την συντεταγμένη x κατά 1 για να μετακινηθεί μια θέση αριστερά.

Για το πλήκτρο K, ελέγχουμε αν ο παίκτης μπορεί να κινηθεί κάτω. Αν μπορεί αυξάνουμε την συντεταγμένη y κατά 1 για να μετακινηθεί μια θέση κάτω.

Για το πλήκτρο I, ελέγχουμε αν ο παίκτης μπορεί να κινηθεί πάνω. Αν μπορεί μειώνουμε την συντεταγμένη y κατά 1 για να μετακινηθεί μια θέση πάνω.

```
311         if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS && !key_j_pressed) {
312             if (player_x - 1 >= 0 && maze[player_y][player_x - 1] == 0) { // κίνηση αριστερά
313                 player_x--;
314             }
315             key_j_pressed = true;
316         }
317         if (glfwGetKey(window, GLFW_KEY_J) == GLFW_RELEASE) {
318             key_j_pressed = false;
319         }
320
321         if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS && !key_k_pressed) {
322             if (player_y + 1 < 10 && maze[player_y + 1][player_x] == 0) { // κίνηση κάτω
323                 player_y++;
324             }
325             key_k_pressed = true;
326         }
327         if (glfwGetKey(window, GLFW_KEY_K) == GLFW_RELEASE) {
328             key_k_pressed = false;
329         }
330
331         if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS && !key_i_pressed) {
332             if (player_y - 1 >= 0 && maze[player_y - 1][player_x] == 0) { // κίνηση πάνω
333                 player_y--;
334             }
335             key_i_pressed = true;
336         }
337         if (glfwGetKey(window, GLFW_KEY_I) == GLFW_RELEASE) {
338             key_i_pressed = false;
339         }
```

Bonus υλοποίηση πέρα από τις ζητούμενες της άσκησης

Προσθέσαμε έναν έλεγχο μέσα στο loop το οποίο ελέγχει αν το τετράγωνο(ο παίκτης) φτάσει στο κελί [7,9] δηλαδή στην έξοδο του λαβύρινθου.

```
342         if (player_x == 9 && player_y == 7) {
343             auto end_time = std::chrono::high_resolution_clock::now();
344             std::chrono::duration<double> elapsed_seconds = end_time - start_time;
345
346             std::cout << "Congratulations, you found the exit!" << std::endl;
347             std::cout << "Time taken: " << elapsed_seconds.count() << " Seconds" << std::endl;
348             break; // Τερματισμός του loop
349         }
```

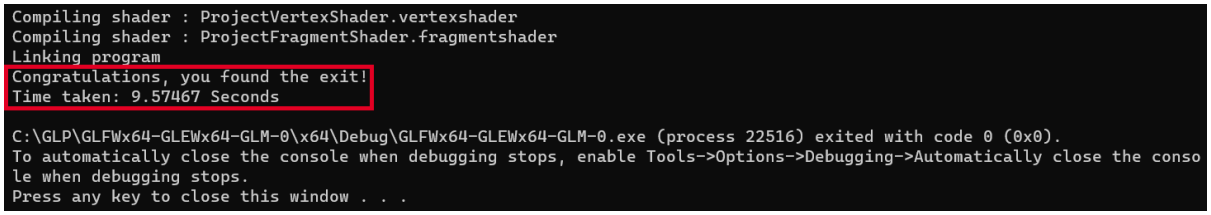
Αν ο παίκτης φτάσει σε αυτό το κελί, το loop τερματίζεται με break και εμφανίζεται μήνυμα στο τερματικό.

Στη συνέχεια, με την χρήση της βιβλιοθήκης <chrono> την οποία κάναμε include στο πρόγραμμα μας `18  #include <chrono>` , προσθέσαμε πριν από το loop τον εξής κώδικα για να καταγράψει τη χρονική στιγμή που ξεκινά το παιχνίδι.

```
287     auto start_time = std::chrono::high_resolution_clock::now();
```

Επίσης, μέσα στον παραπάνω έλεγχο για το αν φτάσει ο παίκτης στην έξοδο του λαβύρινθου, βάλαμε να μετράει την χρονική στιγμή που έφτασε στην έξοδο και με μία

αφαίρεση πόσο χρόνο χρειάστηκε και να εμφανίζει μήνυμα στο τερματικό.



Περιγραφή δυσκολιών υλοποίησης -προβλήματα που συναντήθηκαν

Η μεγαλύτερη δυσκολία ήταν να σκεφτούμε εξ αρχής πως θα υλοποιήσουμε το σχέδιο του λαβύρινθου, χωρίς να πρέπει να γράψουμε τις συντεταγμένες της κάθε κορυφής του κάθε τριγώνου, αλλά φτιάχνοντας το grid του 2Δ πίνακα, και διατρέχοντας τα κελιά του. Ένα πρόβλημα που συναντήσαμε σε αυτό ήταν ότι ο λαβύρινθος εμφανιζόταν flipped (ανεστραμμένος).

Ένα δεύτερο πρόβλημα ήταν η εμφάνιση του τετραγώνου (χαρακτήρα A) στην οθόνη, το οποίο εν τέλει λυνόταν με την δημιουργία των glBindBuffer και της glDrawArrays για το τετράγωνο ξεχωριστά.

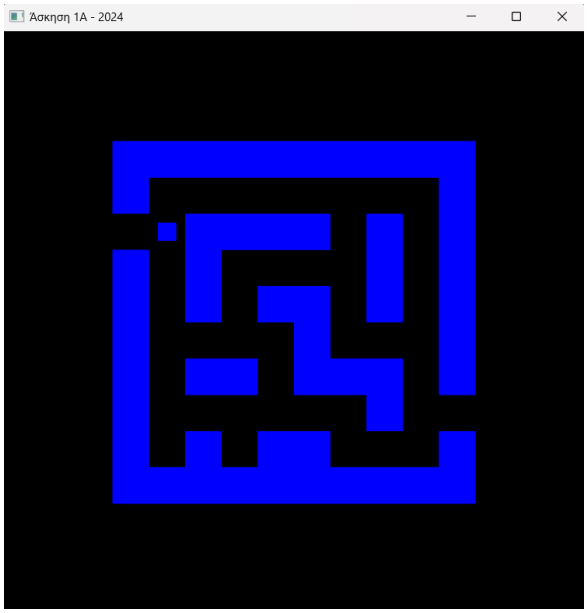
Το τελευταίο πρόβλημα που συναντήσαμε ήταν στην κίνηση του παίκτη, όταν δημιουργήσαμε τους ελέγχους για το αν ο χρήστης έχει πατήσει κάποιο πλήκτρο. Αρχικά, με το πάτημα ενός πλήκτρου (L,J,K, ή I), ο παίκτης μετακινούνταν προς την σωστή κατεύθυνση, αλλά πολλές θέσεις συνεχόμενα μέχρι να βρει τοίχο. Για αυτό τον λόγο ορίσαμε τις λογικές μεταβλητές για το αν το πλήκτρο έχει πατηθεί και στη συνέχεια μέσα στις if ελέγχαμε αν έχει απελευθερωθεί.

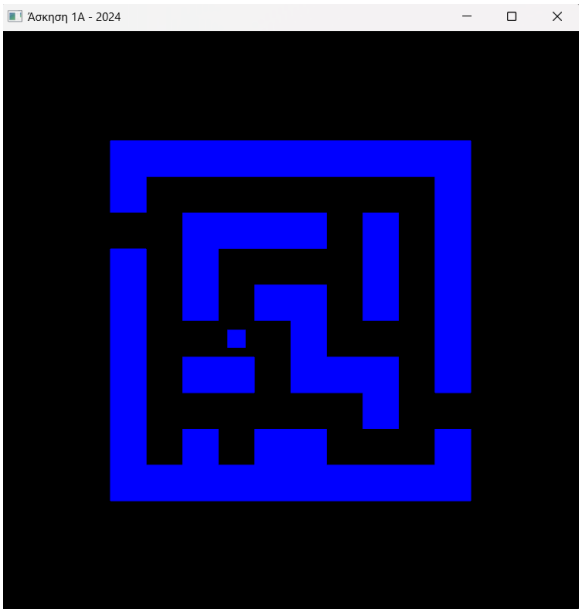
Πληροφορίες σχετικά με την υλοποίηση

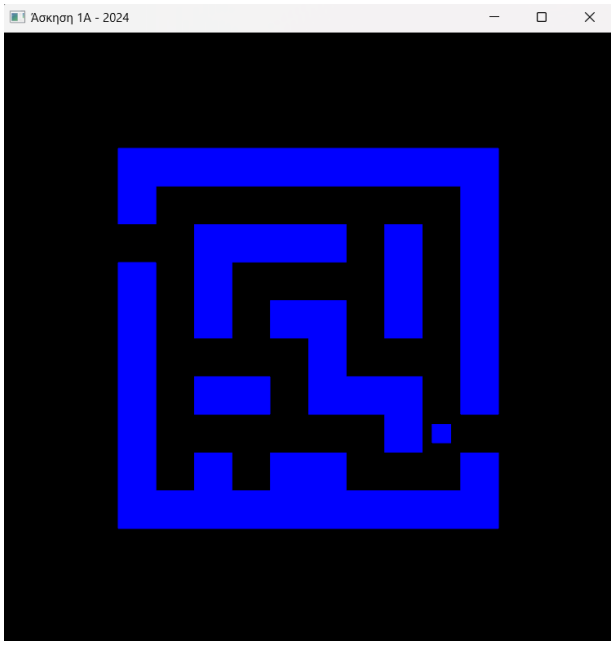
Λειτουργικό σύστημα: Windows 11 Pro Version 23H2

Περιβάλλον: Microsoft Visual Studio Community 2022 (64-bit) - Current Version 17.11.5
Ιδιαιτερότητες στον τρόπο εκτέλεσης - χρήσης: Σύμφωνα με την δική μας προσθήκη που κάναμε, το πρόγραμμα τερματίζει και με το πλήκτρο Q, αλλά και όταν ο παίκτης κινηθεί στην έξοδο του λαβύρινθου.

Στιγμιότυπα







Αναφορές – Πηγές που χρησιμοποιήθηκαν κατά την εκπόνηση της εργασίας.

LearnOpenGL - Hello Triangle
c++ - How to create a grid in OpenGL and drawing it with lines - Stack Overflow
GLFW: Input reference
unicode - UTF-8 Compatibility in C++ - Stack Overflow
Measure execution time of a function in C++ - GeeksforGeeks

Ευχαριστώ για την ανάγνωση.

Παναγιώτης Παρασκευόπουλος
ΑΜ: 2905