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-Γ

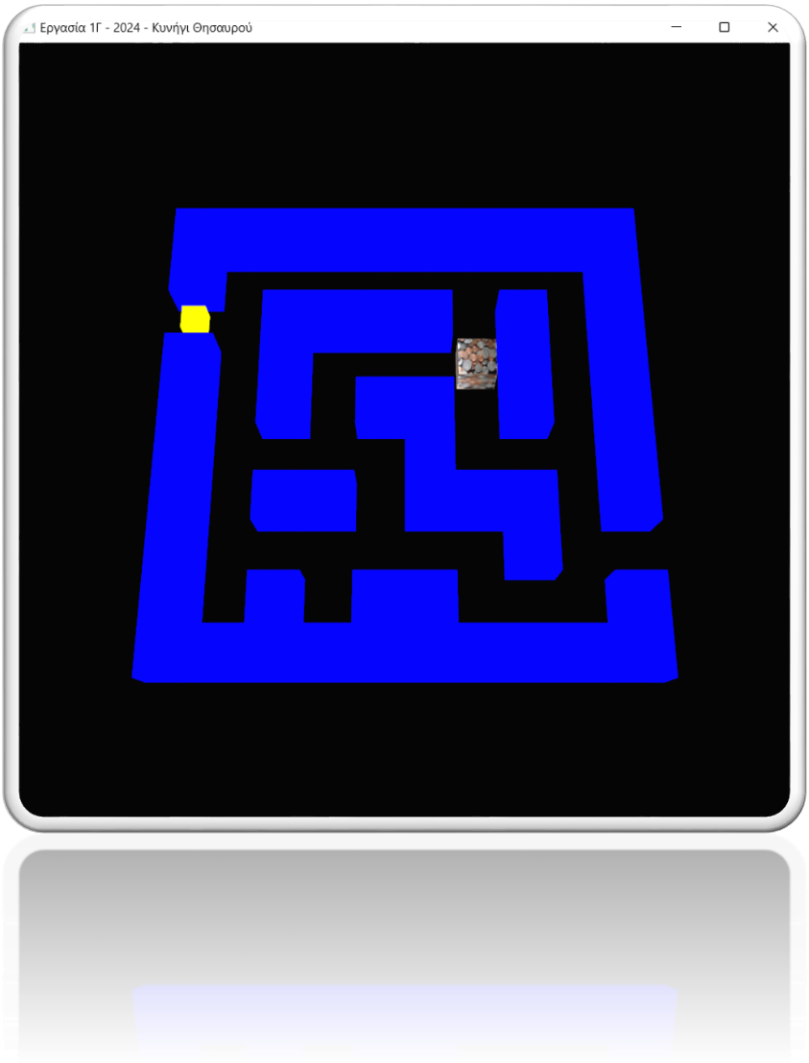
OpenGL 2024-2025

Αναφορά

Παναγιώτης Παρασκευόπουλος

ΑΜ:2905

Περιεχόμενα



Περιγραφή της εργασίας

Η συγκεκριμένη αναφορά βασίζεται στην πρώτη και δεύτερη 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. Στην άσκηση αυτή θα δημιουργήσουμε μια εφαρμογή-παιχνίδι στο οποίο ένας χαρακτήρας θα κυνηγάει «θησαυρούς».

Η άσκηση αυτή έγινε από ένα άτομο. Η υλοποίηση της είναι συνέχεια της υλοποίησης της δεύτερης προγραμματιστικής άσκησης. Στην αναφορά χρησιμοποιώ α’ πληθυντικό για καλύτερη ανάγνωση.

Ερώτημα (i)

Φτιάχνουμε το πρόγραμμα μας να ανοίγει ένα βασικό παράθυρο με μέγεθος 950 x 950 pixels και να έχει τίτλο “Εργασία 1Γ - 2024 - Κυνήγι Θησαυρού”.

```
414     window = glfwCreateWindow(950, 950, u8"Εργασία 1Γ – 2024 – Κυνήγι Θησαυρού", NULL, NULL);
```

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

```
437     // background color black
438     glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

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

```
965     while (glfwGetKey(window, GLFW_KEY_SPACE) != GLFW_PRESS &&
966         glfwWindowShouldClose(window) == 0);
```

Ερώτημα (ii)

•

Η υλοποίηση αυτού του ερωτήματος είναι ίδια με της δεύτερης άσκησης, δηλαδή: Ζωγραφίζουμε τον λαβύρινθο. Ο λαβύρινθος σχηματίζεται ζωγραφίζοντας κύβους μπλε χρώματος, που αντιστοιχούν στα τοιχώματα του λαβύρινθου.

Οπότε ορίζουμε τον maze ως έναν 2Δ πίνακα, που αναπαριστά το λαβύρινθο, ως πλέγμα 10x10 που περιέχει τιμές 0 ή 1. Η τιμή “1” αντιπροσωπεύει τοίχους και η τιμή “0” κενά (μονοπάτι).

```
249     const int maze[10][10] = {
250             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
251             {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
252             {0, 0, 1, 1, 1, 1, 0, 1, 0, 1},
253             {1, 0, 1, 0, 0, 0, 0, 1, 0, 1},
254             {1, 0, 1, 0, 1, 1, 0, 1, 0, 1},
255             {1, 0, 0, 0, 0, 1, 0, 0, 0, 1},
256             {1, 0, 1, 1, 0, 1, 1, 1, 0, 1},
257             {1, 0, 0, 0, 0, 0, 0, 1, 0, 0},
258             {1, 0, 1, 0, 1, 1, 0, 0, 0, 1},
259             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
260     };
```

Δημιουργούμε τις κορυφές (vertices) των τοίχων του λαβύρινθου, που θα χρησιμοποιηθούν για την σχεδίαση του.

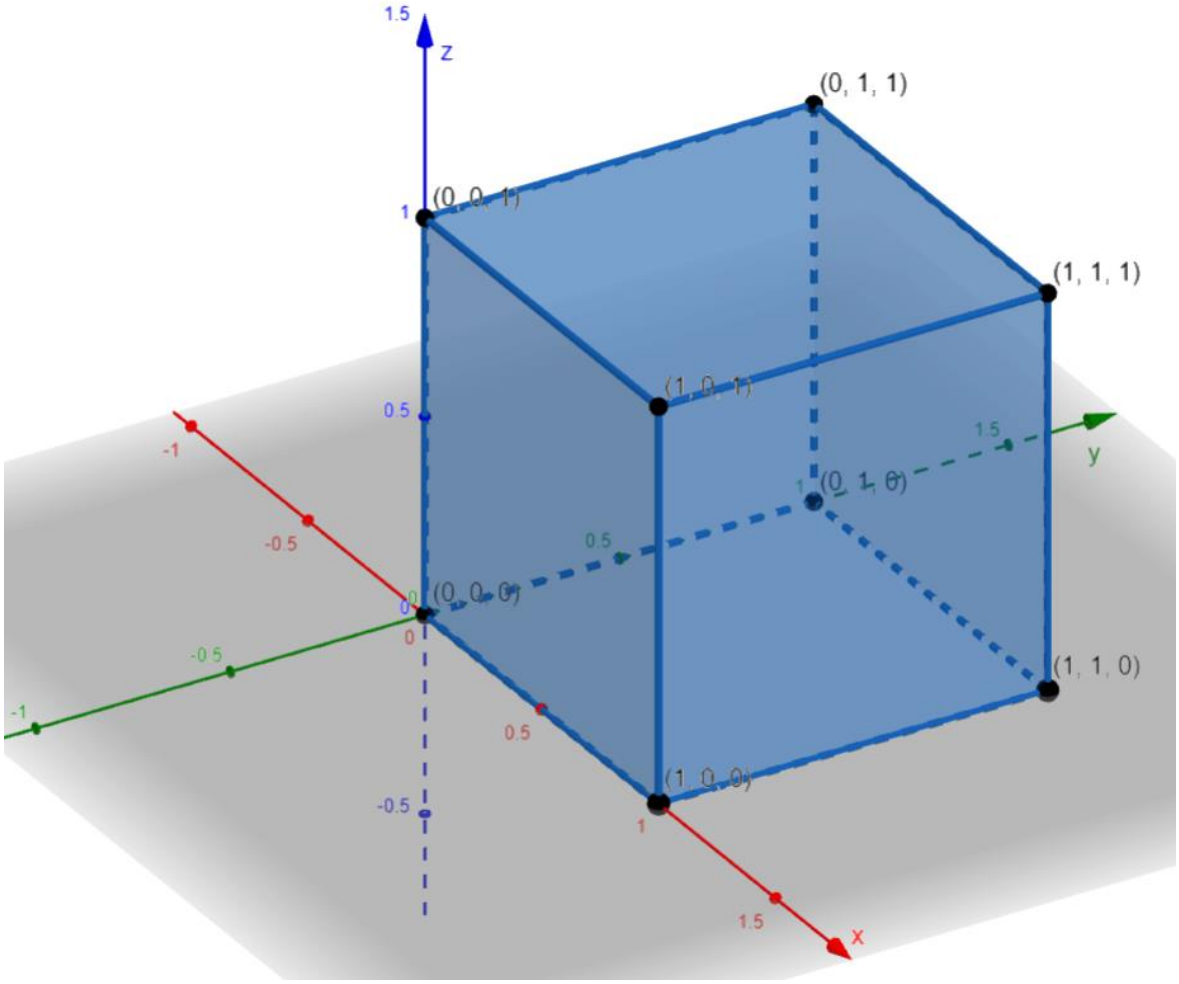
Το μήκος κάθε πλευράς του κύβου να είναι 1.0.
Στη συνέχεια με ένα loop διατρέχουμε τα στοιχεία του πίνακα maze, δηλαδή τις 10 γραμμές και 10 στήλες.
Ελέγχουμε αν το στοιχείο είναι “1” (δηλαδή τοίχος).
Αν είναι, προχωράμε στην δημιουργία των κορυφών. Για κάθε τοίχο του λαβύρινθου δημιουργούμε έναν κύβο. Ο κύβος έχει 6 πλευρές και κάθε πλευρά σχηματίζεται από δύο τρίγωνα. Άρα δημιουργούμε τα 12 τρίγωνα που σχηματίζουν τον κύβο βάζοντας τις κατάλληλες συντεταγμένες για κάθε κορυφή των τριγώνων.

```
std::vector<GLfloat> maze_vertices;

float size = 1.0f;

// Διατρέχουμε τον λαβύρινθο και δημιουργούμε τους κύβους
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (maze[i][j] == 1) {
            // Κεντράρουμε τον λαβύρινθο μετατοπίζοντάς τον ώστε να είναι κεντραρισμένος στο (0,0)
            float x = (j - 5) * size; // Μετατόπιση κατά τον άξονα x (για 10x10, χρησιμοποιούμε -5)
            float y = ((9 - i) - 5) * size; // Αντιστροφή του άξονα y για να αντιστρέψουμε κάθετα τον λαβύρινθο
            float z = 0.0f; // Εδώ μπορούμε να αφήσουμε το z να είναι μηδέν για το κάτω μέρος του κύβου

            // Δημιουργούμε τα 12 τρίγωνα που σχηματίζουν τον κύβο
            // Ο κύβος έχει 6 πλευρές, κάθε πλευρά σχηματίζεται από 2 τρίγωνα
            //
            // Πρόσωπο μπροστά
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);

            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);

            // Πρόσωπο πίσω
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            // Πρόσωπο αριστερά
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);

            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            // Πρόσωπο δεξιά
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);

            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            // Πρόσωπο πάνω
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y + size); maze_vertices.push_back(z);
            maze_vertices.push_back(x);        maze_vertices.push_back(y + size); maze_vertices.push_back(z);

            // Πρόσωπο κάτω
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z);

            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z + size);
            maze_vertices.push_back(x + size); maze_vertices.push_back(y);        maze_vertices.push_back(z);
            maze_vertices.push_back(x);        maze_vertices.push_back(y);        maze_vertices.push_back(z);
        }
    }
}
```

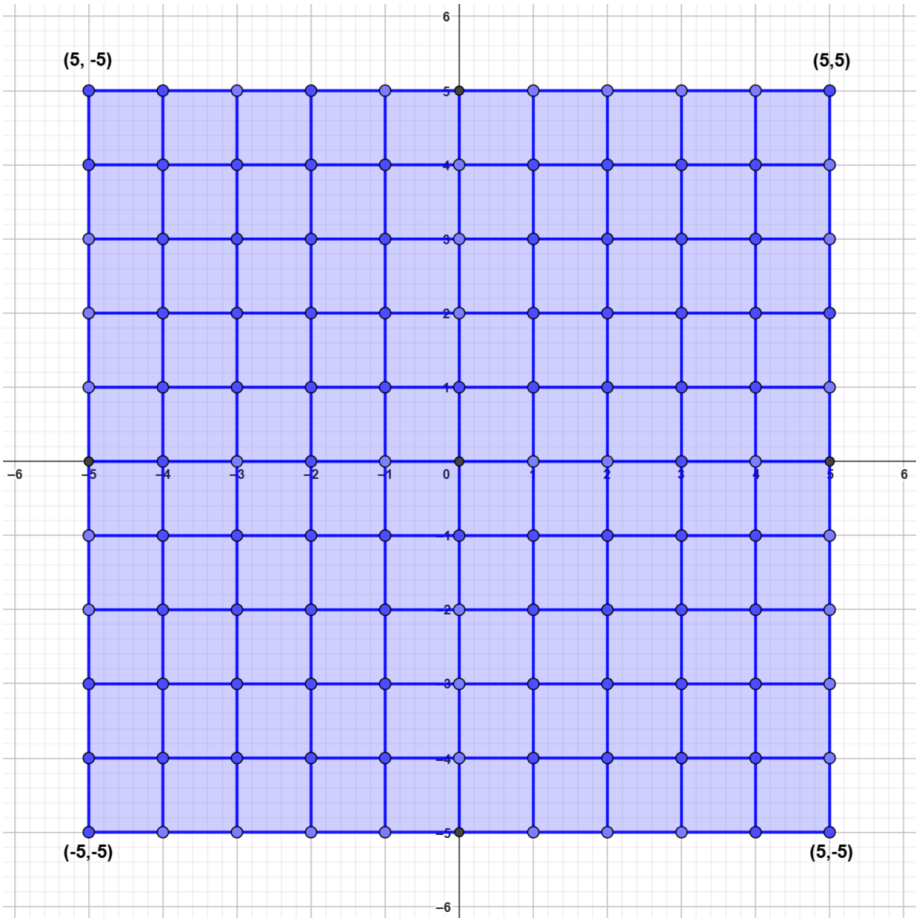
Παίρνουμε για παράδειγμα την μπροστινή(πάνω) πλευρά του κύβου. Κάθε πλευρά αντιστοιχεί σε 6 κορυφές (2 τρίγωνα) που προστίθενται στο maze_vertices. Το πρώτο τρίγωνο σχηματίζεται από τις τρεις πρώτες κορυφές:
Κορυφή 1: (x, y) (κάτω αριστερά)
Κορυφή 2: (x + size, y) (κάτω δεξιά)
Κορυφή 3: (x + size, y + size) (πάνω δεξιά)

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

Από την στιγμή που η πίσω(κάτω) πλευρά του λαβύρινθου «κάθεται» πάνω στο επίπεδο xy (δηλαδή όπου z=0), και κάθε πλευρά έχει μήκος 1, η μπροστινή(πάνω) πλευρά θα βρίσκεται στο z=1. Γι αυτό όλες οι κορυφές των δύο τριγώνων της μπροστινής πλευράς θα είναι (z + size).
Με την ίδια λογική θα σχεδιάσουμε και τις άλλες 5 πλευρές του κύβου τοποθετώντας τις κορυφές στις κατάλληλες θέσεις των σωστών αξόνων.



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



Στη συνέχεια πρέπει να ορίσουμε και το χρώμα του λαβύρινθου να είναι μπλε. Δημιουργούμε έναν πίνακα με 4 στοιχεία που προσδιορίζουν το μπλε χρώμα και τη διαφάνεια (0.0). Μετά, με μια for περνάμε αυτά τα στοιχεία του χρώματος μπλε στις κορυφές των τριγώνων ώστε να χρωματιστεί ο λαβύρινθος.

```
566     // Δεδομένα χρώματος για τα τρίγωνα (μπλε)
567     std::vector<GLfloat> maze_colors;
568
569     // Χρώμα μπλε (RGB = 0, 0, 1, Διαφάνεια = 0.0)
570     GLfloat blueColor[] = { 0.0f, 0.0f, 1.0f, 0.0 };
571
572     // Δημιουργία χρωμάτων για κάθε τρίγωνο του κύβου
573     for (int i = 0; i < maze_vertices.size() / 3; ++i) {
574         maze_colors.push_back(blueColor[0]);
575         maze_colors.push_back(blueColor[1]);
576         maze_colors.push_back(blueColor[2]);
577         maze_colors.push_back(blueColor[3]);
578     }
```

Δημιουργούμε και τα buffer για την αποθήκευση των γεωμετρικών δεδομένων του λαβύρινθου και για τα χρώματα τους.

```
559     GLuint vertexbuffer;
560     glGenBuffers(1, &vertexbuffer);
561     glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
562     glBufferData(GL_ARRAY_BUFFER, maze_vertices.size() * sizeof(GLfloat), maze_vertices.data(), GL_STATIC_DRAW);
```

```
581     GLuint colorbuffer;
582     glGenBuffers(1, &colorbuffer);
583     glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
584     glBufferData(GL_ARRAY_BUFFER, maze_colors.size() * sizeof(GLfloat), maze_colors.data(), GL_STATIC_DRAW);
```

Μέσα στη do κάνουμε bind τα buffer μας για τις συντεταγμένες των

κορυφών(attribute(0)) και των χρωμάτων των κορυφών(attribute(1)) και ενημερώνουμε την glDrawArrays για τον σχεδιασμό των τριγώνων του λαβύρινθου στην οθόνη.

```
862     // Attribute buffers for maze
863     // 1rst attribute buffer : vertices for maze
864     glEnableVertexAttribArray(0);
865     glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
866     glVertexAttribPointer(
867         0,
868         3,
869         GL_FLOAT,
870         GL_FALSE,
871         0,
872         (void*)0
873     );
874
875     // 2nd attribute buffer : colors for maze
876     glEnableVertexAttribArray(1);
877     glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
878     glVertexAttribPointer(
879         1,
880         4,                                  // size
881         GL_FLOAT,
882         GL_FALSE,
883         0,
884         (void*)0
885     );
886
887     // Draw triangles
888     glDrawArrays(GL_TRIANGLES, 0, maze_vertices.size() / 3);
889     glDisableVertexAttribArray(0);
```

-

Δημιουργούμε ένα μικρότερο κίτρινο κύβο (χαρακτήρας A), αντιστοιχεί σε έναν κινούμενο χαρακτήρα που διασχίζει τον λαβύρινθο.
Φτιάχνουμε τη συνάρτηση update_square_A_vertices, η οποία υπολογίζει τις κορυφές ενός κύβου βασισμένο στη θέση του παίκτη.

```
589     auto update_cube_A_vertices = [&](int x, int y) -> std::vector<GLfloat> {
590         float cell_size = 1.0f;
591         float center_x = (x - 5) * cell_size + 0.25f;
592         float center_y = ((9 - y) - 5) * cell_size + 0.25f;
593         float center_z = 0.25f;
594
595         // Το μέγεθος του κύβου
596         float size = 0.5f;  // Μισό του μήκους της ακμής του κύβου (κύβος 0.5
```

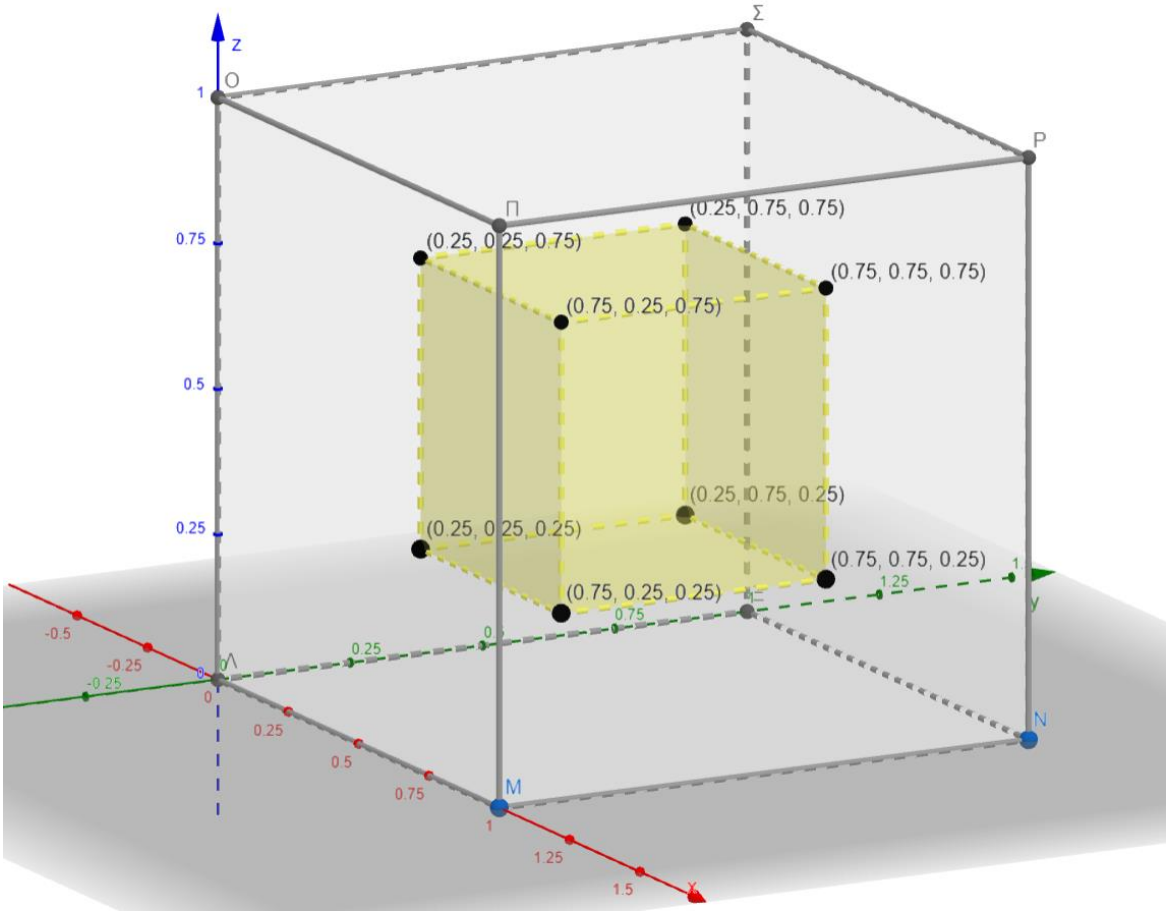
Ορίζουμε το cell_size ως 1.0f και αντιπροσωπεύει το μέγεθος κάθε κελιού στο grid. Ορίζουμε το size ως 0.5f και αντιπροσωπεύει το μήκος κάθε πλευράς του κύβου(χαρακτήρα A). Ορίζουμε τα center_x, center_y και center_z τα οποία υπολογίζουν τη θέση του κέντρου του κελιού με βάση τις συντεταγμένες (x, y, z).

Η συνάρτηση επιστρέφει έναν std::vector<GLfloat>, ο οποίος περιέχει τις συντεταγμένες για τις 36 κορυφές του κύβου. Είναι ίδια λογική με την σχεδίαση των κύβων για τα κελιά του λαβύρινθου με την διαφορά ότι στον χαρακτήρα A το μήκος των πλευρών του κύβου είναι 0.5.

```
598     return {
599         // Εμπρόσθια πλευρά (6 κορυφές, 2 τρίγωνα)
600         center_x,        center_y,        center_z + size,  // κάτω αριστερά
601         center_x + size, center_y,        center_z + size,  // κάτω δεξιά
602         center_x + size, center_y + size, center_z + size,  // πάνω δεξιά
603
604         center_x,        center_y,        center_z + size,  // κάτω αριστερά
605         center_x + size, center_y + size, center_z + size,  // πάνω δεξιά
606         center_x,        center_y + size, center_z + size,  // πάνω αριστερά
607
608         // Πίσω πλευρά (6 κορυφές, 2 τρίγωνα)
609         center_x,        center_y,        center_z,         // κάτω αριστερά
610         center_x + size, center_y,        center_z,         // κάτω δεξιά
611         center_x + size, center_y + size, center_z,         // πάνω δεξιά
612
613         center_x,        center_y,        center_z,         // κάτω αριστερά
614         center_x + size, center_y + size, center_z,         // πάνω δεξιά
615         center_x,        center_y + size, center_z,         // πάνω αριστερά
616
617         // Δεξιά πλευρά (6 κορυφές, 2 τρίγωνα)
618         center_x + size, center_y,        center_z + size,  // εμπρόσθιο κάτω δεξιά
619         center_x + size, center_y + size, center_z + size,  // εμπρόσθιο πάνω δεξιά
620         center_x + size, center_y + size, center_z,         // πίσω πάνω δεξιά
621
622         center_x + size, center_y,        center_z + size,  // εμπρόσθιο κάτω δεξιά
623         center_x + size, center_y + size, center_z,         // πίσω πάνω δεξιά
624         center_x + size, center_y,        center_z,         // πίσω κάτω δεξιά
625
626         // Αριστερή πλευρά (6 κορυφές, 2 τρίγωνα)
627         center_x,        center_y,        center_z + size,  // εμπρόσθιο κάτω αριστερά
628         center_x,        center_y + size, center_z + size,  // εμπρόσθιο πάνω αριστερά
629         center_x,        center_y + size, center_z,         // πίσω πάνω αριστερά
630
631         center_x,        center_y,        center_z + size,  // εμπρόσθιο κάτω αριστερά
632         center_x,        center_y + size, center_z,         // πίσω πάνω αριστερά
633         center_x,        center_y,        center_z,         // πίσω κάτω αριστερά
634
635         // Άνω πλευρά (6 κορυφές, 2 τρίγωνα)
636         center_x,        center_y + size, center_z + size,  // εμπρόσθιο πάνω αριστερά
637         center_x + size, center_y + size, center_z + size,  // εμπρόσθιο πάνω δεξιά
638         center_x + size, center_y + size, center_z,         // πίσω πάνω δεξιά
639
640         center_x,        center_y + size, center_z + size,  // εμπρόσθιο πάνω αριστερά
641         center_x + size, center_y + size, center_z,         // πίσω πάνω δεξιά
642         center_x,        center_y + size, center_z,         // πίσω πάνω αριστερά
643
644         // Κάτω πλευρά (6 κορυφές, 2 τρίγωνα)
645         center_x,        center_y,        center_z + size,  // εμπρόσθιο κάτω αριστερά
646         center_x + size, center_y,        center_z + size,  // εμπρόσθιο κάτω δεξιά
647         center_x + size, center_y,        center_z,         // πίσω κάτω δεξιά
648
649         center_x,        center_y,        center_z + size,  // εμπρόσθιο κάτω αριστερά
650         center_x + size, center_y,        center_z,         // πίσω κάτω δεξιά
651         center_x,        center_y,        center_z          // πίσω κάτω αριστερά
652     };
```

Παράδειγμα μπροστινής πλευράς κύβου:
Πρώτο τρίγωνο: κάτω αριστερή, κάτω δεξιά, πάνω δεξιά κορυφή.
Δεύτερο τρίγωνο: κάτω αριστερή, πάνω δεξιά, πάνω αριστερή κορυφή .

Χρειάστηκε να προσθέσουμε 0.25 στα center_x, center_y και center_z για να κεντραριστεί ο κύβος με μήκος πλευράς 0.5 στο κέντρο του κελιού που έχει πλευρά με μήκος 1.0.



Η cube_A_vertices παίρνει την τιμή που επιστρέφει η update_cube_A_vertices (player_x, player_y), δηλαδή τις αρχικές κορυφές του κύβου που αντιστοιχούν στη θέση του παίκτη.

```
657         std::vector<GLfloat> cube_A_vertices = update_cube_A_vertices(player_x, player_y);
```

Ορίζουμε και τον πίνακα με τα χρώματα των κορυφών το κύβου(χαρακτήρα Α) με το χρώμα κίτρινο και διαφάνεια 0.0.

```
660     std::vector<GLfloat> cube_A_colors = {
661         1.0f, 1.0f, 0.0f, 0.0f,
662         1.0f, 1.0f, 0.0f, 0.0f,
663         1.0f, 1.0f, 0.0f, 0.0f,
664
665         1.0f, 1.0f, 0.0f, 0.0f,
666         1.0f, 1.0f, 0.0f, 0.0f,
667         1.0f, 1.0f, 0.0f, 0.0f,
668
669         1.0f, 1.0f, 0.0f, 0.0f,
670         1.0f, 1.0f, 0.0f, 0.0f,
671         1.0f, 1.0f, 0.0f, 0.0f,
672
673         1.0f, 1.0f, 0.0f, 0.0f,
674         1.0f, 1.0f, 0.0f, 0.0f,
675         1.0f, 1.0f, 0.0f, 0.0f,
676
677         1.0f, 1.0f, 0.0f, 0.0f,
678         1.0f, 1.0f, 0.0f, 0.0f,
679         1.0f, 1.0f, 0.0f, 0.0f,
680
681         1.0f, 1.0f, 0.0f, 0.0f,
682         1.0f, 1.0f, 0.0f, 0.0f,
683         1.0f, 1.0f, 0.0f, 0.0f,
684
685         1.0f, 1.0f, 0.0f, 0.0f,
686         1.0f, 1.0f, 0.0f, 0.0f,
687         1.0f, 1.0f, 0.0f, 0.0f,
688
689         1.0f, 1.0f, 0.0f, 0.0f,
690         1.0f, 1.0f, 0.0f, 0.0f,
691         1.0f, 1.0f, 0.0f, 0.0f,
692
693         1.0f, 1.0f, 0.0f, 0.0f,
694         1.0f, 1.0f, 0.0f, 0.0f,
695         1.0f, 1.0f, 0.0f, 0.0f,
696
697         1.0f, 1.0f, 0.0f, 0.0f,
698         1.0f, 1.0f, 0.0f, 0.0f,
699         1.0f, 1.0f, 0.0f, 0.0f,
700
701         1.0f, 1.0f, 0.0f, 0.0f,
702         1.0f, 1.0f, 0.0f, 0.0f,
703         1.0f, 1.0f, 0.0f, 0.0f,
704
705         1.0f, 1.0f, 0.0f, 0.0f,
706         1.0f, 1.0f, 0.0f, 0.0f,
707         1.0f, 1.0f, 0.0f, 0.0f
708     };
```

Δημιουργούμε και τα buffers της OpenGL για την αποθήκευση των γεωμετρικών δεδομένων του κύβου και των χρωμάτων του.

```
717     GLuint cube_vertexbuffer;
718     glGenBuffers(1, &cube_vertexbuffer);
719     glBindBuffer(GL_ARRAY_BUFFER, cube_vertexbuffer);
720     glBufferData(GL_ARRAY_BUFFER, cube_A_vertices.size() * sizeof(GLfloat), cube_A_vertices.data(), GL_STATIC_DRAW);
721
722     // Δημιουργία του buffer για το χρώμα του κύβου
723     GLuint cube_colorbuffer;
724     glGenBuffers(1, &cube_colorbuffer);
725     glBindBuffer(GL_ARRAY_BUFFER, cube_colorbuffer);
726     glBufferData(GL_ARRAY_BUFFER, cube_A_colors.size() * sizeof(GLfloat), cube_A_colors.data(), GL_STATIC_DRAW);
```

Όπως και στην περίπτωση του λαβύρινθου, μέσα στη do κάνουμε bind τα buffer μας για τις συντεταγμένες των κορυφών(attribute(0)) και των χρωμάτων των κορυφών(attribute(1)) και ενημερώνουμε την glDrawArrays για τον σχεδιασμό των τριγώνων του κύβου(χαρακτήρα A) στην οθόνη.

```
891     // Attribute buffers for cube A
892     // 1rst attribute buffer : vertices for cube A
893     glEnableVertexAttribArray(0);
894     glBindBuffer(GL_ARRAY_BUFFER, cube_vertexbuffer);
895     glVertexAttribPointer(
896         0,
897         3,
898         GL_FLOAT,
899         GL_FALSE,
900         0,
901         (void*)0
902     );
903
904     // 2nd attribute buffer : colors for cube A
905     glEnableVertexAttribArray(1);
906     glBindBuffer(GL_ARRAY_BUFFER, cube_colorbuffer);
907     glVertexAttribPointer(
908         1,
909         4,                                  // size
910         GL_FLOAT,
911         GL_FALSE,
912         0,
913         (void*)0
914     );
915
916     // Draw triangles
917     glDrawArrays(GL_TRIANGLES, 0, 36);
```

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

```
855     // Ενημέρωση κορυφών του κυβου A
856     cube_A_vertices = update_cube_A_vertices(player_x, player_y);
```

```
858     // Ενημέρωση του buffer με τις νέες κορυφές
859     glBindBuffer(GL_ARRAY_BUFFER, cube_vertexbuffer);
860     glBufferData(GL_ARRAY_BUFFER, cube_A_vertices.size() * sizeof(GLfloat), cube_A_vertices.data(), GL_STATIC_DRAW);
```

Κίνηση του παίκτη(χαρακτήρα A):

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.

```
739     bool key_l_pressed = false;
740     bool key_j_pressed = false;
741     bool key_k_pressed = false;
742     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 SPACE) ελέγχουμε αν το πλήκτρο είναι πατημένο ή όχι.

Αναλύουμε για το πλήκτρο “L”

```
766         // Ανάγνωση πλήκτρων για την κίνηση
767         if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS && !key_l_pressed) {
768             if (player_x + 1 < 10 && maze[player_y][player_x + 1] == 0) { // κίνηση δεξιά
769                 player_x++;
770             }
771             else if (player_x + 1 >= 10) {
772                 player_x = 0;
773                 player_y = 2;
774             }
775
776             key_l_pressed = true; // Σημείωσε ότι το πλήκτρο είναι τώρα πατημένο
777         }
778         if (glfwGetKey(window, GLFW_KEY_L) == GLFW_RELEASE) {
779             key_l_pressed = false; // Επαναφορά όταν το πλήκτρο απελευθερωθεί
780         }
```

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.

```
782         if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS && !key_j_pressed) {
783             if (player_x - 1 >= 0 && maze[player_y][player_x - 1] == 0) { // κίνηση αριστερά
784                 player_x--;
785             }
786             else if (player_x - 1 < 0) {
787                 player_x = 9;
788                 player_y = 7;
789             }
790 
791             key_j_pressed = true;
792         }
793         if (glfwGetKey(window, GLFW_KEY_J) == GLFW_RELEASE) {
794             key_j_pressed = false;
795         }
796 
797         if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS && !key_k_pressed) {
798             if (player_y + 1 < 10 && maze[player_y + 1][player_x] == 0) { // κίνηση κάτω
799                 player_y++;
800             }
801             key_k_pressed = true;
802         }
803         if (glfwGetKey(window, GLFW_KEY_K) == GLFW_RELEASE) {
804             key_k_pressed = false;
805         }
806 
807         if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS && !key_i_pressed) {
808             if (player_y - 1 >= 0 && maze[player_y - 1][player_x] == 0) { // κίνηση πάνω
809                 player_y--;
810             }
811             key_i_pressed = true;
812         }
813         if (glfwGetKey(window, GLFW_KEY_I) == GLFW_RELEASE) {
814             key_i_pressed = false;
815         }
```

Μέσα στον έλεγχο για το πάτημα των πλήκτρων 'L' και 'J' βάζουμε και μία else if για να ελέγξουμε αν η κίνηση του παίκτη είναι εκτός των ορίων του λαβύρινθου και αν ισχύει τοποθετούμε τον χαρακτήρα A στην είσοδο(κελί [2,0]) ή στην έξοδο(κελί[7,9]) αντίστοιχα του λαβύρινθου.

Ορίζουμε την αρχική θέση του παίκτη στον λαβύρινθο.

```
261     int player_x = 0; // στήλη 0
262     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 κελί της τρίτης γραμμής και πρώτης στήλης του λαβύρινθου.

Ερώτημα(iii)

Δημιουργούμε το σχέδιο του θησαυρού με παρόμοια λογική όπως αυτή του χαρακτήρα A.

```cpp
std::vector<GLfloat> generate_treasure_vertices(int x, int y, float shrink_factor = 1.0f) {
    float cell_size = 1.0f;
    float center_x = (x - 5) * cell_size + 0.5f; // Κέντρο κυττάρου (x)
    float center_y = ((9 - y) - 5) * cell_size + 0.5f; // Κέντρο κυττάρου (y)
    float center_z = 0.5f; // Σταθερό ύψος

    float size = 0.8f * shrink_factor; // Μέγεθος πλευράς με shrinking
    float half_size = size / 2.0f;     // Μισό μέγεθος για συμμετρική συρρίκνωση

    return {
        // Εμπρόσθια πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x - half_size, center_y - half_size, center_z + half_size,  // κάτω αριστερά
        center_x + half_size, center_y - half_size, center_z + half_size,  // κάτω δεξιά
        center_x + half_size, center_y + half_size, center_z + half_size,  // πάνω δεξιά

        center_x - half_size, center_y - half_size, center_z + half_size,  // κάτω αριστερά
        center_x + half_size, center_y + half_size, center_z + half_size,  // πάνω δεξιά
        center_x - half_size, center_y + half_size, center_z + half_size,  // πάνω αριστερά

        // Πίσω πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x - half_size, center_y - half_size, center_z - half_size,  // κάτω αριστερά
        center_x + half_size, center_y - half_size, center_z - half_size,  // κάτω δεξιά
        center_x + half_size, center_y + half_size, center_z - half_size,  // πάνω δεξιά

        center_x - half_size, center_y - half_size, center_z - half_size,  // κάτω αριστερά
        center_x + half_size, center_y + half_size, center_z - half_size,  // πάνω δεξιά
        center_x - half_size, center_y + half_size, center_z - half_size,  // πάνω αριστερά

        // Αριστερή πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x - half_size, center_y - half_size, center_z - half_size,  // πίσω κάτω
        center_x - half_size, center_y - half_size, center_z + half_size,  // εμπρός κάτω
        center_x - half_size, center_y + half_size, center_z + half_size,  // εμπρός πάνω

        center_x - half_size, center_y - half_size, center_z - half_size,  // πίσω κάτω
        center_x - half_size, center_y + half_size, center_z + half_size,  // εμπρός πάνω
        center_x - half_size, center_y + half_size, center_z - half_size,  // πίσω πάνω

        // Δεξιά πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x + half_size, center_y - half_size, center_z - half_size,  // πίσω κάτω
        center_x + half_size, center_y - half_size, center_z + half_size,  // εμπρός κάτω
        center_x + half_size, center_y + half_size, center_z + half_size,  // εμπρός πάνω

        center_x + half_size, center_y - half_size, center_z - half_size,  // πίσω κάτω
        center_x + half_size, center_y + half_size, center_z + half_size,  // εμπρός πάνω
        center_x + half_size, center_y + half_size, center_z - half_size,  // πίσω πάνω

        // Κάτω πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x - half_size, center_y - half_size, center_z - half_size,  // πίσω αριστερά
        center_x + half_size, center_y - half_size, center_z - half_size,  // πίσω δεξιά
        center_x + half_size, center_y - half_size, center_z + half_size,  // εμπρός δεξιά

        center_x - half_size, center_y - half_size, center_z - half_size,  // πίσω αριστερά
        center_x + half_size, center_y - half_size, center_z + half_size,  // εμπρός δεξιά
        center_x - half_size, center_y - half_size, center_z + half_size,  // εμπρός αριστερά

        // Πάνω πλευρά (6 κορυφές, 2 τρίγωνα)
        center_x - half_size, center_y + half_size, center_z - half_size,  // πίσω αριστερά
        center_x + half_size, center_y + half_size, center_z - half_size,  // πίσω δεξιά
        center_x + half_size, center_y + half_size, center_z + half_size,  // εμπρός δεξιά

        center_x - half_size, center_y + half_size, center_z - half_size,  // πίσω αριστερά
        center_x + half_size, center_y + half_size, center_z + half_size,  // εμπρός δεξιά
        center_x - half_size, center_y + half_size, center_z + half_size   // εμπρός αριστερά
    };
}
```

Η διαφορά σε σχέση με τον κύβο(χαρακτήρα A) είναι ότι ορίζουμε τις συντεταγμένες των κορυφών του θησαυρού έτσι ώστε όταν γίνει συρρίκνωση στη συνέχεια, να γίνει προς το κέντρο του. Για αυτό το λόγο έχουμε κινηθεί στο κέντρο του κελιού και ορίζουμε κατά 0.4 ή -0.4 ως προς τους x, y, z άξονες (επειδή το μήκος πλευράς του θησαυρού είναι 0.8). Έχουμε ορίσει και τον shrink_factor για να γίνει η συρρίκνωση στη συνέχεια.

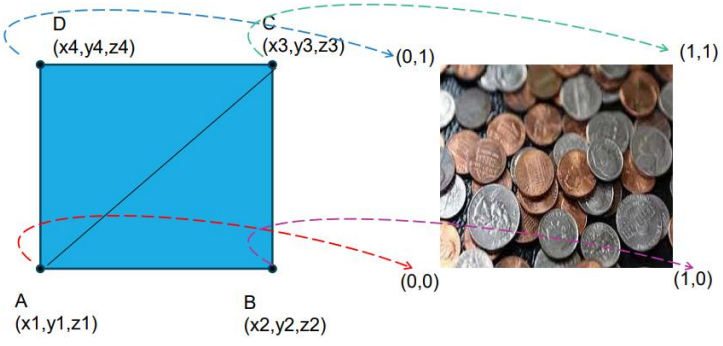
Έπειτα, ορίζουμε τις UV συντεταγμένες των κορυφών που θα πάρει το texture που θα του δώσουμε.

```cpp
std::vector<GLfloat> generate_treasure_uvs() {
    return {
        // Μπροστινή πλευρά
            0.0f, 0.0f,  // κάτω αριστερά
            1.0f, 0.0f,  // κάτω δεξιά
            1.0f, 1.0f,  // πάνω δεξιά

            0.0f, 0.0f,  // κάτω αριστερά
            1.0f, 1.0f,  // πάνω δεξιά
            0.0f, 1.0f,  // πάνω αριστερά

            0.0f, 0.0f,
            1.0f, 0.0f,
            1.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 1.0f,
            0.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 0.0f,
            1.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 1.0f,
            0.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 0.0f,
            1.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 1.0f,
            0.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 0.0f,
            1.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 1.0f,
            0.0f, 1.0f

            0.0f, 0.0f,
            1.0f, 0.0f,
            1.0f, 1.0f,

            0.0f, 0.0f,
            1.0f, 1.0f,
            0.0f, 1.0f
    };
}
```

Για κάθε πλευρά του κύβου(θησαυρού) ορίζουμε τις κορυφές των δύο τριγώνων ως:
(0,0) κάτω αριστερή γωνία
(1,0) κάτω δεξιά γωνία
(1,1) πάνω δεξιά γωνία

(0,0) κάτω αριστερή γωνία
(0,1) πάνω αριστερή γωνία
(1,1) πάνω δεξιά γωνία

όπως ήταν και στο παράδειγμα του εργαστηρίου



Δημιουργούμε και τα buffers της OpenGL για την αποθήκευση των γεωμετρικών δεδομένων του θησαυρού και των uv συντεταγμένων του.

```
728     GLuint treasure_vertexbuffer;
729     glGenBuffers(1, &treasure_vertexbuffer);
730     glBindBuffer(GL_ARRAY_BUFFER, treasure_vertexbuffer);
731     glBufferData(GL_ARRAY_BUFFER, treasure_vertices.size() * sizeof(GLfloat), treasure_vertices.data(), GL_STATIC_DRAW);
732
733     GLuint treasure_uvbuffer;
734     glGenBuffers(1, &treasure_uvbuffer);
735     glBindBuffer(GL_ARRAY_BUFFER, treasure_uvbuffer);
736     glBufferData(GL_ARRAY_BUFFER, treasure_uvs.size() * sizeof(GLfloat), treasure_uvs.data(), GL_STATIC_DRAW);
```

Μέσα στη do κάνουμε bind τα buffer μας για τις συντεταγμένες των κορυφών και των uv συντεταγμένων.

```
924         glEnableVertexAttribArray(0);
925         glBindBuffer(GL_ARRAY_BUFFER, treasure_vertexbuffer);
926         glVertexAttribPointer(
927             0,
928             3,                                  // size
929             GL_FLOAT,
930             GL_FALSE,
931             0,
932             (void*)0
933         );
934
935         glEnableVertexAttribArray(2);
936         glBindBuffer(GL_ARRAY_BUFFER, treasure_uvbuffer);
937         glVertexAttribPointer(
938             2,
939             2,                                  // size
940             GL_FLOAT,
941             GL_FALSE,
942             0,
943             (void*)0
944         );
```

Τώρα θέλουμε να δώσουμε υφή (texture) στον θησαυρό και όχι χρώμα όπως στον λαβύρινθο και στον παίκτη.
Δημιουργούμε μία νέα υφή την αποθηκεύουμε στο textureID και την κάνουμε bind.

```
467     GLuint textureID;
468     glGenTextures(1, &textureID);
469     glBindTexture(GL_TEXTURE_2D, textureID);
```

Φορτώνουμε την εικόνα coins και επιστρέφουμε τα δεδομένα της εικόνας, πλάτος (width), το ύψος (height) και τον αριθμό των καναλιών (nrChannels, π.χ. RGB = 3 κανάλια) της εικόνας.

```
471     int width, height, nrChannels;
472     stbi_set_flip_vertically_on_load(true); // Αναστροφή εικόνας
473     unsigned char* data = stbi_load("coins.jpg", &width, &height, &nrChannels, 0);
```

Η συνάρτηση stbi_set_flip_vertically_on_load της βιβλιοθήκης stb_image ρυθμίζει την εικόνα να αναστρέφεται κατακόρυφα κατά την φόρτωσή της, γιατί η OpenGL φορτώνει τις εικόνες με το κάτω μέρος προς τα πάνω.
Δημιουργούμε την υφή στην OpenGL με τα δεδομένα της εικόνας στις δυο διαστάσεις.

```
475     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
476     glGenerateMipmap(GL_TEXTURE_2D);
```

Δημιουργούμε επίσης mipmaps για την υφή, τα οποία χρησιμοποιούνται για να βελτιώσουν την ποιότητα της υφής όταν αυτή εμφανίζεται σε μικρότερες διαστάσεις στην οθόνη.
Επίσης, ρυθμίζουμε τις παραμέτρους της υφής που καθορίζουν πώς θα

συμπεριφέρεται η υφή κατά την επαναλαμβανόμενη εμφάνιση (wrap) και κατά την κλίμακα (filtering).

```
479         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // Επανάληψη σε άξονα S
480         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT); // Επανάληψη σε άξονα T
481         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR); // Μείωση με mipmaps
482         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Μεγέθυνση με γραμμική φιλτράρισμα
```

GL_TEXTURE_WRAP_S και GL_TEXTURE_WRAP_T: Καθορίζουν τι θα συμβεί όταν η υφή υπερβεί τα όρια του αντικειμένου. Το GL_REPEAT κάνει την υφή να επαναλαμβάνεται.
GL_TEXTURE_MIN_FILTER: Καθορίζει το φίλτρο που χρησιμοποιείται όταν η υφή εμφανίζεται μικρότερη από την αρχική της διάσταση. Το GL_LINEAR_MIPMAP_LINEAR σημαίνει ότι χρησιμοποιούνται mipmaps και γραμμικό φίλτρο για την ελαχιστοποίηση της απόστασης.
GL_TEXTURE_MAG_FILTER: Καθορίζει το φίλτρο για όταν η υφή εμφανίζεται μεγαλύτερη. Το GL_LINEAR σημαίνει ότι θα χρησιμοποιείται γραμμικό φίλτρο.

Αφού έχουμε φορτώσει την υφή στο πρόγραμμά μας πρέπει να υλοποιήσουμε τα Vertex και Fragment Shaders ώστε να την εφαρμόσουμε κατάλληλα.
Ο Vertex shader παίρνει σαν input τις uv συντεταγμένες και το χρώμα και τις περνάει στον Fragment shader.

```
1   #version 330 core
2   layout(location = 0) in vec3 vertexPosition_modelspace;
3   layout(location = 1) in vec4 vertexColor;
4   layout(location = 2) in vec2 vertexUV;
5
6   out vec2 UV;
7   out vec4 fragmentColor;
8
9   uniform mat4 MVP;
10
11  void main() {
12      gl_Position = MVP * vec4(vertexPosition_modelspace, 1);
13      UV = vertexUV;
14      fragmentColor = vertexColor;
15  }
```

Ο Fragment shader παίρνει σαν input τις uv συντεταγμένες και το output θα είναι το χρώμα που αντιστοιχίζεται σε κάθε pixel.

```
1   #version 330 core
2   in vec2 UV;
3   in vec4 fragmentColor;
4   out vec4 color;
5
6   uniform sampler2D myTexture;
7   uniform int color_choice;
8
9   void main() {
10      if (color_choice == 0){
11          color = fragmentColor;
12      } else {
13          color = texture(myTexture, UV);
14      }
15
16  }
```

Έχουμε ορίσει την color_choice και ανάλογα με την τιμή της εφαρμόζουμε χρώμα ή υφή στο κάθε pixel.
Αν color_choice είναι 0, τα αντικείμενα σχεδιάζονται με το προκαθορισμένο χρώμα

τους.
Αν το color_choice είναι 1, τα αντικείμενα καλύπτονται από την υφή myTexture.
Μέσα στο πρόγραμμα μας έχουμε ορίσει την choose_color που καλεί την color_choice η οποία ελέγχει την επιλογή χρώματος ή υφής για το αντικείμενο.

```
449         int choose_color = glGetUniformLocation(programID, "color_choice");
450         choose_color = 0;
```

Την αρχικοποιούμε στο 0 για να έχει προεπιλεγμένη την επιλογή χρώματος.

Μέσα στην do πριν ενεργοποιήσουμε και συνδέσουμε τα buffer για τα δεδομένα του λαβύρινθου, του παίκτη και των χρωμάτων τους, στέλνουμε την τιμή 0 στην color_choice για να καθορίσει ότι ο λαβύρινθος και ο παίκτης θα πάρουν τα χρώματα που τους δίνουμε.

```
857         glUniform1i(glGetUniformLocation(programID, "color_choice"), 0);
```

Ενώ πιο κάτω, πριν ενεργοποιήσουμε και συνδέσουμε τα buffer για τα δεδομένα του θησαυρού και των uv συντεταγμένων στέλνουμε την τιμή 1 στην color_choice για να καθορίσει ότι ο θησαυρός θα πάρει το texture που δίνουμε και το ενεργοποιούμε.

```
919         glUniform1i(glGetUniformLocation(programID, "color_choice"), 1);
920         glActiveTexture(GL_TEXTURE0);
921         glBindTexture(GL_TEXTURE_2D, textureID);
922         glUniform1i(glGetUniformLocation(programID, "myTexture"), 0);
```

Ερώτημα (iv)

Αρχικά μέσα στην do ελέγχουμε αν υπάρχει επαφή του παίκτη με τον θησαυρό. Αν υπάρχει , ξεκινάει η συρρίκνωση και αποθηκεύουμε τη θέση του θησαυρού όταν έγινε η επαφή.

```
818         if (player_x == treasure_x && player_y == treasure_y && !is_shrinking) {
819             sndPlaySound(TEXT("coinSound.wav"), SND_ASYNC);
820
821             is_shrinking = true;  // Έναρξη συρρίκνωσης
822             shrink_timer = glfwGetTime();
823
824             // Αποθήκευση τρέχουσας θέσης θησαυρού
825             prev_treasure_x = treasure_x;
826             prev_treasure_y = treasure_y;
827         }
```

Στη συνέχεια υλοποιούμε τι γίνεται όταν η συρρίκνωση είναι ενεργή.

```
828         if (is_shrinking) {
829             // Υπολογισμός ποσοστού συρρίκνωσης
830             float elapsed_shrink_time = glfwGetTime() - shrink_timer;
831             float shrink_factor = 1.0f - (elapsed_shrink_time / shrink_duration);
832             if (shrink_factor < 0.5f) shrink_factor = 0.5f; // Ελάχιστο μέγεθος στο 50%
833
834             // Δημιουργία νέων κορυφών με βάση το `shrink_factor`
835             treasure_vertices = generate_treasure_vertices(treasure_x, treasure_y, shrink_factor);
836             glBindBuffer(GL_ARRAY_BUFFER, treasure_vertexbuffer);
837             glBufferData(GL_ARRAY_BUFFER, treasure_vertices.size() * sizeof(GLfloat), treasure_vertices.data(), GL_STATIC_DRAW);
838
839             // Έλεγχος αν ολοκληρώθηκε η συρρίκνωση
840             if (elapsed_shrink_time >= shrink_duration) {
841                 is_shrinking = false;      // Τέλος συρρίκνωσης
842                 treasure_collected = true; // Ο θησαυρός θεωρείται συλλεγμένος
843
844                 // Εξαφάνιση θησαυρού
845                 treasure_x = -1;
846                 treasure_y = -1;
847             }
848             // Μετακίνηση παίκτη στο κελί του θησαυρού
849             player_x = prev_treasure_x;
850             player_y = prev_treasure_y;
851         }
```

Υπολογίζουμε τον χρόνο που έχει περάσει από την έναρξη της συρρίκνωσης. Μετά υπολογίζουμε το ποσοστό συρρίκνωσης. Το shrink_factor αρχικά είναι 1.0 και μειώνεται όσο περνάει ο χρόνος. Ορίζουμε το ελάχιστο μέγεθος του θησαυρού στο 50% της αρχικής του διάστασης και δημιουργούμε νέες κορυφές για τον θησαυρό με βάση το τρέχον ποσοστό συρρίκνωσης. Δένουμε και ενημερώνουμε τα buffer με τα νέα δεδομένα.
Έπειτα ελέγχουμε αν η διάρκεια της συρρίκνωσης έχει ολοκληρωθεί και τη σταματάμε τη διαδικασία. Εξαφανίζουμε τον θησαυρό αφού έχει συλλεχθεί και τοποθετούμε τον παίκτη σε αυτό το κελί.

Τέλος, κάνουμε και έναν ακόμη έλεγχο για να εξασφαλίσουμε ότι ο θησαυρός είτε θα εμφανίζεται περιοδικά είτε θα εμφανίζεται ξανά αφού συλλεχθεί από τον παίκτη.

```
947             treasure_timer = glfwGetTime();
948             if ((treasure_timer - previous_time >= treasure_visibility_duration || treasure_collected) && !is_shrinking) {
949                 treasure_collected = false;        // Επαναφορά κατάστασης θησαυρού
950                 update_treasure_position();        // Νέα θέση θησαυρού
951                 treasure_vertices = generate_treasure_vertices(treasure_x, treasure_y, 1.0f); // Επαναφορά μεγέθους
952                 previous_time = treasure_timer;    // Ενημέρωση χρονικού σημείου
953             }
```

Ελέγχουμε αν έχει περάσει αρκετός χρόνος από την τελευταία εμφάνιση του θησαυρού ή αν ο θησαυρός έχει ήδη συλλεχθεί και εξασφαλίζουμε ότι ο θησαυρός δεν θα εμφανιστεί ξανά κατά τη διάρκεια της φάσης συρρίκνωσης. Καλούμε την update_treasure_position, επαναφέρουμε τον θησαυρό σε μέγεθος 1.0 και ενημερώνουμε και τον χρόνο.
Η υλοποίηση της update_treasure_position είναι να επιλέγει έναν αριθμό μεταξύ 0 και 9 (συντεταγμένες στον πίνακα maze) αφού ελέγξει να μην είναι τοίχος του λαβύρινθου και να μην είναι στην ίδια θέση με τον παίκτη.

```
270 void update_treasure_position() {
271     do {
272         treasure_x = rand() % 10;
273         treasure_y = rand() % 10;
274     } while (maze[treasure_y][treasure_x] == 1 || (treasure_x == player_x && treasure_y == player_y));
275 }
```

Έχουμε εφαρμόσει και την srand για να παίρνουμε κάθε φορά τυχαίους αριθμούς.

```
710         srand(static_cast<unsigned int>(time(0)));
```

Ερώτημα (v)

Η υλοποίηση της κάμερας είναι ίδια ακριβώς με της προηγούμενης άσκησης, δεν έχει υποστεί καμία αλλαγή εκτός από την προσθήκη να μετακινείται η κάμερα στον άξονα x (panning) με τα πλήκτρα ‘g’ και ‘h’ και στον άξονα y με τα πλήκτρα και ‘t’ και ‘b’.

```
65      float panningStep = 0.1f; // Ο ρυθμός μετακίνησης για το panning
```

```
117     // Μετακίνηση στον άξονα X (panning)
118     if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS) {
119         cameraPosition.x -= panningStep; // Μετακίνηση προς τα αριστερά
120         cameraCenterPoint.x -= panningStep; // Ενημέρωση του σημείου που κοιτάζει η κάμερα
121     }
122     if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS) {
123         cameraPosition.x += panningStep; // Μετακίνηση προς τα δεξιά
124         cameraCenterPoint.x += panningStep; // Ενημέρωση του σημείου που κοιτάζει η κάμερα
125     }
126
127     // Μετακίνηση στον άξονα Y (panning)
128     if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS) {
129         cameraPosition.y += panningStep; // Μετακίνηση προς τα πάνω
130         cameraCenterPoint.y += panningStep; // Ενημέρωση του σημείου που κοιτάζει η κάμερα
131     }
132     if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS) {
133         cameraPosition.y -= panningStep; // Μετακίνηση προς τα κάτω
134         cameraCenterPoint.y -= panningStep; // Ενημέρωση του σημείου που κοιτάζει η κάμερα
135     }
```

Bonus υλοποίηση

Η μοναδική επιπλέον υλοποίηση που έγινε, ήταν η προσθήκη ήχου κατά την επαφή του παίκτη με τον θησαυρό.
Κάναμε include τις βιβλιοθήκες για τον ήχο

```
23  #include <windows.h>
24  #include <mmsystem.h>
25  #pragma comment(lib, "Winmm.lib")
```

Και προσθέσαμε μέσα στον έλεγχο επαφής παίκτη με θησαυρό, την sndPlaySound για την αναπαραγωγή του ήχου από WAV αρχείο.

```
819                 sndPlaySound(TEXT("coinSound.wav"), SND_ASYNC);
```

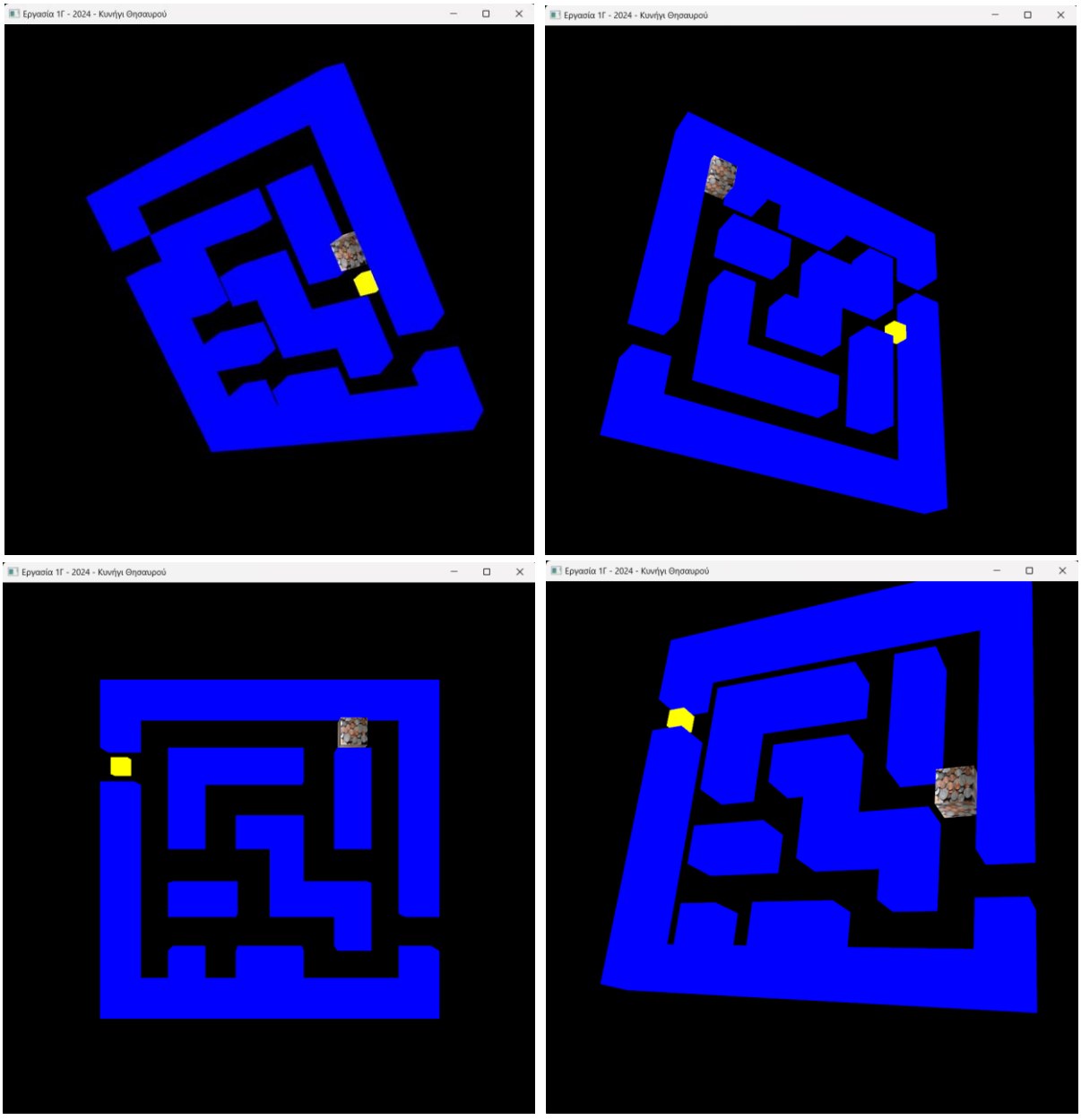
Περιγραφή δυσκολιών υλοποίησης -προβλήματα που συναντήθηκαν

Έχοντας υλοποιήσει την πρώτη και δεύτερη εργαστηριακή άσκηση και βασισμένοι πάνω σε αυτήν, δεν συναντήθηκαν δυσκολίες στην κατασκευή του σχήματος του θησαυρού. Μεγάλη δυσκολία υπήρξε στην εμφάνιση της υφής πάνω στον θησαυρό που μετά από πολλές ώρες προσπάθειας και διαφόρων τροποποιήσεων πραγματοποιήθηκε. Επίσης μικρή δυσκολία συναντήθηκε στην υλοποίηση της συρρίκνωσης αλλά και ταυτόχρονης αλλαγής θέσης του θησαυρού και εμφάνισης του παίκτη στο κελί που έγινε η επαφή.

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

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
stackoverflow-is-there-a-function-to-calculate-this-unit-vector-in-glm
https://www.opengl-tutorial.org/beginners-tutorials/tutorial-4-a-colored-cube/
https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
https://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/
https://github.com/nothings/stb/blob/master/stb_image.h
https://www.softwaretestinghelp.com/random-number-generator-cpp/

Ευχαριστώ για την ανάγνωση.

Παναγιώτης Παρασκευόπουλος
ΑΜ: 2905