ΓΡΑΦΙΚΑ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΣΥΣΤΗΜΑΤΑ ΑΛΛΗΛΕΠΙΔΡΑΣΗΣ

Προγραμματιστική Άσκηση 1-C.

Team

Γκόβαρης Χρήστος-Γρηγόριος
Σπανού Μαρία

Χρονοδιάγραμμα Εργαστηριακής Άσκησης 1-C

Έκδοση	Ημερομηνία	Progress
1.0	18/11/2024	Ανακοίνωση εργαστηριακής άσκησης 1-C
2.0	27/11/2024	Υλοποίηση των ερωτημάτων (i) και (ii)
2.1	30/11/2024	Υλοποίηση του ερωτήματος (iii) και bonus (α)
2.2	30/11/2024	Υλοποίηση του ερωτήματος (iv)
2.3	30/11/2024	Υλοποίηση του ερωτήματος (v)
2.4	1/12/2024	Υλοποίηση του ερωτήματος (vi)
2.5	2/12/2024	Υλοποίηση των bonus ερωτημάτων
3.0	3/12/2024	Υλοποίηση Report εργαστηριακής άσκησης

Ανάλυση Χαρακτηριστικών

Η εργασία υλοποιήθηκε σε ένα μηχάνημα (Lenovo Ideapad 3), με τα εξής χαρακτηριστικά πυρήνα:

- AMD Ryzen 7 (7730U)
- 8 CPU cores
- 16 Threads
- Boost Clock up to 4.5GHz
- Base Clock 2.0GHz
- CPU Socket FP6
- Intergraded Graphics (Radeon Graphics)

Επιπλέον, η **εργασία 1-C** υλοποιήθηκε σε **Windows 11 Home**, χρησιμοποιώντας το **Visual Studio 2022 (x64)**.

Λειτουργία Ομάδας / Ανάλυση Ερωτημάτων

Η συνεργασία της ομάδας ήταν υποδειγματική, εξασφαλίζοντας την επιτυχή υλοποίηση της Άσκησης 1-C. Τα ερωτήματα (i) και (ii) προσαρμόστηκαν από την Άσκηση 1-B, στην οποία είχαν υλοποιηθεί με επιτυχία. Στο ερώτημα (iii), επιλύθηκε ένα πρόβλημα που προέκυψε με την εφαρμογή του texture (MeshLab). Στο ερώτημα (iv), ολοκληρώθηκε επιτυχώς η μείωση του σχήματος του treasure χαρακτήρα. Στο ερώτημα (v), προσαρμόστηκε η κίνηση της κάμερας από την Άσκηση 1-B και προστέθηκε panning. Από τα bonus ερωτήματα, υλοποιήθηκε μόνο το (α).

2.0 Υλοποίηση των ερωτημάτων (i) και (ii)

Για το ερώτημα (i):

υλοποιήθηκε με τον ίδιο τρόπο όπως στην Προγραμματιστική Άσκηση 1Β

Για το ερώτημα (ii):

υλοποιήθηκε με τον ίδιο τρόπο όπως στην Προγραμματιστική Άσκηση 1Β

2.1 Υλοποίηση του ερωτήματος (iii) και bonus (α)

Στο παρόν ερώτημα, μας ζητήθηκε να υλοποιήσουμε την περιοδική εμφάνιση του αντικειμένου Β (θησαυρού) στον λαβύρινθο, όπως περιγράφεται στην εκφώνηση. Ο θησαυρός εμφανίζεται σε τυχαία θέση που δεν συμπίπτει με τη θέση του χαρακτήρα Α ή με τοίχο, παραμένει για συγκεκριμένο χρονικό διάστημα και στη συνέχεια μετακινείται σε νέα τυχαία θέση. Το κέντρο του θησαυρού ευθυγραμμίζεται με το κέντρο του κύβου του λαβύρινθου όπου εμφανίζεται. Ο θησαυρός είναι κύβος με μήκος πλευράς 0.8, στον οποίο εφαρμόστηκε η υφή **coins.jpg** με υπολογισμό των uv συντεταγμένων. Πιο συγκεκριμένα, έγιναν οι εξής αλλαγές:

Στο αρχείο **Source-1C.cpp**, υλοποιήσαμε την εντολή

```
int activeTextureIndex = 0;


void placeTreasure(int maze[10][10], float* treasureX, float* treasureY, float* treasureZ, GLfloat character_verticesB[], float* characterX, float* characterY, GLuint vertexbuffer3,) {
    static double lastPlacementTime = 0.0;
    static bool isFirstAppearance = true;
    const double treasureDuration = 5.0;
    double currentTime = glfwGetTime();
    static bool textureSelected = false;


 if (!isFirstAppearance && (currentTime - lastPlacementTime) < treasureDuration) {
        return;
    }


    std::cout << "Placing treasure...\n";
    int row, col;
    int attempts = 0;
    const int maxAttempts = 100;
    bool isOnCharacter;


    do {
        row = rand() % 10;
        col = rand() % 10;
        attempts++;
        if (attempts >= maxAttempts) {
            std::cout << "Could not find a valid position for treasure!\n";
            return;
        }
        *treasureX = col - 5.0f + 0.1f;
        *treasureY = 5.0f - row - 0.1f;
```

```
    isOnCharacter = (fabs((*treasureX) - *characterX) < 0.8f &&
                     fabs((*treasureY) - *characterY) < 0.8f);
} while (maze[row][col] != 0 || isOnCharacter);

if (!textureSelected) {
    activeTextureIndex = rand() % 3;
    std::cout << "Active texture index set to: " << activeTextureIndex << std::endl;
    textureSelected = true;
}

GLfloat tempVertices[] = {
    *treasureX, *treasureY, *treasureZ,
    *treasureX, *treasureY - 0.8f, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ,
    *treasureX, *treasureY, *treasureZ,
    *treasureX + 0.8f, *treasureY, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ,
    *treasureX, *treasureY, *treasureZ - 0.8f,
    *treasureX, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX, *treasureY, *treasureZ - 0.8f,
    *treasureX + 0.8f, *treasureY, *treasureZ - 0.8f,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX + 0.8f, *treasureY, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX, *treasureY, *treasureZ,
    *treasureX, *treasureY - 0.8f, *treasureZ,
    *treasureX, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX, *treasureY, *treasureZ,
    *treasureX, *treasureY, *treasureZ - 0.8f,
    *treasureX, *treasureY - 0.8f, *treasureZ - 0.8f,
    *treasureX + 0.8f, *treasureY, *treasureZ - 0.8f,
    *treasureX, *treasureY - 0.8f, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ,
    *treasureX + 0.8f, *treasureY - 0.8f, *treasureZ - 0.8f,
};
std::copy(std::begin(tempVertices), std::end(tempVertices), character_verticesB);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3);
glBufferData(GL_ARRAY_BUFFER, sizeof(tempVertices), tempVertices, GL_STATIC_DRAW);
lastPlacementTime = glfwGetTime();
isFirstAppearance = false;
textureSelected = false;   };
```

Σύμφωνα με τις οποίες, αρχικοποιήσαμε τυχαία τις συντεταγμένες **x**, **y** και **z** του θησαυρού B, ώστε να καθορίσουμε τη θέση του στον λαβύρινθο. Οι συντεταγμένες αυτές υπολογίζονται τυχαία, με βάση έναν πίνακα που αναπαριστά τη δομή του λαβύρινθου (**maze**). Στη συνέχεια, δημιουργήσαμε έναν πίνακα **GLfloat tempVertices[]**, ο οποίος περιέχει τις συντεταγμένες των κορυφών για την απόδοση του θησαυρού στον τρισδιάστατο χώρο. Για την εμφάνιση του θησαυρού, χρησιμοποιήσαμε **Vertex Buffer Objects (VBOs)**, τα οποία ενημερώνονται δυναμικά κάθε φορά που αλλάζει η θέση του θησαυρού, και συνδέονται με το **OpenGL pipeline** μέσω του **glBindBuffer** και **glBufferData**. Το αντικείμενο αποθηκεύεται στο **buffer** και τυπώνεται στο παράθυρο με βάση τις νέες συντεταγμένες. Τέλος, ο θησαυρός B εμφανίζεται με μία από τρεις τυχαίες υφές, η οποία επιλέγεται κατά την τοποθέτησή του μέσω του **activeTextureIndex**, που ορίζεται τυχαία κάθε φορά. Η διαδικασία διασφαλίζει ότι ο θησαυρός δεν εμφανίζεται σε θέση που καταλαμβάνεται από τον χαρακτήρα A ή από τοίχο του λαβύρινθου, ενώ ανανεώνεται περιοδικά μετά από συγκεκριμένο χρονικό διάστημα.

Επίσης, οι uv συντεταγμένες καθορίζονται

```
GLfloat g_uv_buffer_data[] = {
    // Front Face
    0.000000, 0.333333, 0.333333, 0.000000, 0.333333, 0.333333, 0.000000, 0.666667, 0.333333, 0.333333, 0.333333, 0.666667,
    // Right Face
    0.333333, 0.333333, 0.666667, 0.000000, 0.666667, 0.333333, 0.333333, 0.666667, 0.666667, 0.333333, 0.666667, 0.666667,
    // Back Face
    0.333333, 1.000000, 0.000000, 0.666667, 0.333333, 0.666667, 0.666667, 1.000000, 0.333333, 0.666667, 0.666667, 0.666667,
    // Left Face
    0.000000, 0.000000, 0.000000, 0.333333, 0.333333, 0.000000, 0.333333, 0.333333, 0.000000, 1.000000, 0.333333, 1.000000,
    // Top Face
    0.666667, 0.666667, 1.000000, 0.666667, 1.000000, 1.000000, 0.666667, 1.000000, 0.666667, 0.666667, 1.000000, 0.666667,
    // Bottom Face
    0.666667, 0.000000, 1.000000, 0.000000, 1.000000, 0.333333, 0.666667, 0.333333, 0.666667, 0.000000, 1.000000, 0.333333,
};
```

σύμφωνα με τον παραπάνω πίνακα **GLfloat g_uv_buffer_data[]**, για την εφαρμογή υφής στις έξι επιφάνειες του κύβου που αναπαριστά τον θησαυρό B. Κάθε σετ τιμών UV αντιστοιχεί σε μία από τις επιφάνειες: την μπροστινή, τη δεξιά, την πίσω, την αριστερή, την πάνω και την κάτω. Οι συντεταγμένες αυτές χαρτογραφούν τμήματα της υφής σε κάθε επιφάνεια, εξασφαλίζοντας ότι η υφή προβάλλεται σωστά σε όλες τις πλευρές του κύβου. Η δομή τους ακολουθεί τη μορφή **(u, v)**, όπου **u** είναι η οριζόντια και **v** η κάθετη συντεταγμένη μέσα στην υφή, επιτρέποντας την ακριβή τοποθέτησή της στο αντικείμενο.

Επιπροσθέτως, υλοποιήσαμε τον κώδικα

```
GLuint vertexbuffer3;
glGenBuffers(1, &vertexbuffer3);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3);
glBufferData(GL_ARRAY_BUFFER, sizeof(charachter_verticesB), charachter_verticesB, GL_STATIC_DRAW);

GLuint uvbuffer;
glGenBuffers(1, &uvbuffer);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_uv_buffer_data), g_uv_buffer_data, GL_STATIC_DRAW);
```

σύμφωνα με τον οποίο, αρχικοποιούνται δύο **Vertex Buffer Objects (VBOs)** για την αποθήκευση και αποστολή δεδομένων κορυφών και UV συντεταγμένων στο GPU. Ο πρώτος buffer, **vertexbuffer3**, δημιουργείται με την κλήση της **glGenBuffers**, δεσμεύεται με **glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3)**, και γεμίζεται με τα δεδομένα κορυφών από τον πίνακα **character_verticesB** μέσω της **glBufferData**. Αυτό επιτρέπει στο OpenGL να αποθηκεύσει τις συντεταγμένες των κορυφών του χαρακτήρα ή του θησαυρού στον buffer για χρήση κατά την απόδοση. Ο δεύτερος buffer, **uvbuffer**, δημιουργείται με την ίδια διαδικασία, αλλά αυτή τη φορά αποθηκεύει τις UV συντεταγμένες από τον πίνακα **g_uv_buffer_data**. Η **glBufferData** μεταφέρει τα δεδομένα UV στον buffer, ώστε οι συντεταγμένες αυτές να χρησιμοποιηθούν για τη χαρτογράφηση υφής στις επιφάνειες του αντικειμένου. Και οι δύο buffers είναι απαραίτητοι για τη σωστή απεικόνιση του τρισδιάστατου αντικειμένου με τις αντίστοιχες υφές.

Για τις τρεις υφές (coins, gold και silver), υλοποιήσαμε τον κώδικα

```
GLuint textureIDs[3];
std::string textureFiles[3] = { "coins.jpg", "silver.jpg", "gold.jpg" };
int width, height, nrChannels;
for (int i = 0; i < 3; ++i) {
    unsigned char* data = stbi_load(textureFiles[i].c_str(), &width, &height, &nrChannels, 0);
    if (data) {
        std::cout << "Image loaded successfully!" << std::endl;
        std::cout << "Width: " << width << ", Height: " << height << ", Channels: " << nrChannels << std::endl;
    }
    else {
        std::cerr << "Failed to load texture: " << textureFiles[i] << std::endl;
    }
    glGenTextures(1, &textureIDs[i]);
    glBindTexture(GL_TEXTURE_2D, textureIDs[i]);
```

```
    GLenum format = (nrChannels == 4) ? GL_RGBA : GL_RGB;
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    stbi_image_free(data);
    if (textureIDs[i] == 0) {
        std::cerr << "Texture creation failed!" << std::endl;
    }
    else {
        std::cout << "Texture ID: " << textureIDs[i] << " loaded successfully!" << std::endl;
    }
}
```

Σύμφωνα με τον οποίο, αρχικοποιούνται τρεις υφές για τον θησαυρό Β, χρησιμοποιώντας έναν πίνακα **textureFiles** που περιέχει τα αρχεία εικόνας ("coins.jpg", "silver.jpg", "gold.jpg"). Για κάθε υφή, φορτώνεται το αντίστοιχο αρχείο εικόνας μέσω της **stbi_load**, η οποία επιστρέφει τα δεδομένα της εικόνας μαζί με τις διαστάσεις της (**width**, **height**) και τον αριθμό καναλιών (**nrChannels**). Αν η φόρτωση είναι επιτυχής, εμφανίζεται μήνυμα επιβεβαίωσης με τις λεπτομέρειες της εικόνας, διαφορετικά καταγράφεται σφάλμα. Στη συνέχεια, δημιουργείται ένα **Texture Object** με την **glGenTextures**, δεσμεύεται με την **glBindTexture**, και τα δεδομένα της εικόνας μεταφέρονται στη GPU μέσω της **glTexImage2D**, με τη μορφή να καθορίζεται από τον αριθμό καναλιών (GL_RGB ή GL_RGBA). Ορίζονται οι παράμετροι υφής με τις κλήσεις **glTexParameteri**, για την επανάληψη της υφής (**GL_REPEAT**) και τη γραμμική δειγματοληψία (**GL_LINEAR**) κατά την μεγέθυνση ή σμίκρυνση της εικόνας. Τέλος, τα δεδομένα εικόνας αποδεσμεύονται με την **stbi_image_free**, ενώ καταγράφεται επιτυχία ή αποτυχία δημιουργίας της υφής μέσω του **textureIDs[i]**. Αυτή η διαδικασία διασφαλίζει ότι οι τρεις διαφορετικές υφές είναι έτοιμες για εφαρμογή στον θησαυρό Β κατά την απόδοση.

Για την σύνδεση των υφών, υλοποιήσαμε τον κώδικα

```
glUniform1i(glGetUniformLocation(programID, "texture1"), 0);         // GL_TEXTURE0
glUniform1i(glGetUniformLocation(programID, "texture2"), 1);         // GL_TEXTURE1
glUniform1i(glGetUniformLocation(programID, "texture3"), 2);         // GL_TEXTURE2
```

σύμφωνα με τον οποίο, οι κλήσεις **glUniform1i** χρησιμοποιούνται για τη σύνδεση των υφών με τα αντίστοιχα **sampler2D** uniforms στο shader πρόγραμμα. Κάθε **glGetUniformLocation** επιστρέφει τη θέση μνήμης των uniforms ("texture1", "texture2", "texture3") μέσα στο shader, ενώ η τιμή που ορίζεται (0, 1, 2) αντιστοιχεί σε μονάδες υφής (**GL_TEXTURE0**, **GL_TEXTURE1**, **GL_TEXTURE2**). Αυτό επιτρέπει στον shader να

κατανοεί ποια υφή να χρησιμοποιήσει κατά την απόδοση, με βάση την ενεργή μονάδα υφής που έχει συνδεθεί μέσω της **glActiveTexture** και της **glBindTexture**. Έτσι, εξασφαλίζεται ότι οι τρεις διαφορετικές υφές είναι σωστά αντιστοιχισμένες και έτοιμες για χρήση στη σκηνή.

Για την ρύθμιση των **attribute buffers**, υλοποιήσαμε

```
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3);
glVertexAttribPointer(
   0,
   3,
   GL_FLOAT,
   GL_FALSE,
   0,
   (void)0
);
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glVertexAttribPointer(
   2,
   2,
   GL_FLOAT,
   GL_FALSE,
   0,
   (void)0
);
glUniform1i(useTextureID, 1);
glUniform1i(activeTextureUniform, activeTextureIndex + 1);
glActiveTexture(GL_TEXTURE0 + activeTextureIndex);
glBindTexture(GL_TEXTURE_2D, textureIDs[activeTextureIndex]);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(2);
```

ο οποίος διαχειρίζεται τη ρύθμιση τους και την απόδοση του αντικειμένου. Αρχικά, απενεργοποιούνται τα attribute arrays, και στη συνέχεια ενεργοποιούνται τα απαραίτητα. Το **vertexbuffer3** συνδέεται με το attribute 0 για τις κορυφές, ενώ το **uvbuffer** συνδέεται με το attribute 2 για τις UV συντεταγμένες. Τα δεδομένα κορυφών και UV συντεταγμένων περνούν στους shaders μέσω της **glVertexAttribPointer**. Στη συνέχεια, οι uniforms ρυθμίζονται για τη χρήση της υφής και την επιλογή της ενεργής υφής, με την **glUniform1i** να ορίζει το **activeTextureIndex** και το **glActiveTexture** να επιλέγει τη σωστή μονάδα υφής. Η ενεργή υφή συνδέεται με το **glBindTexture**, και το αντικείμενο αποδίδεται μέσω της **glDrawArrays**. Τέλος, τα attribute arrays απενεργοποιούνται ξανά για να διασφαλιστεί η καθαρή κατάσταση του pipeline.

Τέλος, η κλήση στην **main**, έχει γίνει ως εξής:

```
placeTreasure(maze, &treasureX, &treasureY, &treasureZ, charachter_verticesB, &x, &y, vertexbuffer3);
```

Επίσης, στον **P1BFragmentShader.fragmentshader**, προσθέσαμε τον ακόλουθο κώδικα

```
in vec2 UV;

uniform sampler2D texture1;

uniform sampler2D texture2;

uniform sampler2D texture3;


uniform int activeTexture;

uniform int useTexture;

vec4 baseColor;

  if (useTexture == 1) {

    if (activeTexture == 1) {

      baseColor = texture(texture1, UV).rgba;

    } else if (activeTexture == 2) {

      baseColor = texture(texture2, UV).rgba;

    } else if (activeTexture == 3) {

      baseColor = texture(texture3, UV).rgba;

    } else {

      baseColor = vec4(1.0, 0.0, 0.0, 1.0);

    }

  } else {

    baseColor = fragmentColor;

  }
```

σύμφωνα ο οποίος επιλέγει ποια υφή θα εφαρμοστεί σε ένα αντικείμενο, ανάλογα με την τιμή του uniform **activeTexture**. Εάν η μεταβλητή **useTexture** είναι 1, ελέγχεται η τιμή του **activeTexture** για να επιλεγεί η σωστή υφή από τα **texture1**, **texture2**, ή **texture3**. Ανάλογα με την τιμή του **activeTexture**, η κατάλληλη υφή φορτώνεται χρησιμοποιώντας τη συνάρτηση **texture()** και οι UV συντεταγμένες εφαρμόζονται για να ανακτηθεί το χρώμα της υφής στην αντίστοιχη θέση. Αν η τιμή του **activeTexture** δεν είναι 1, 2 ή 3, επιστρέφεται ένα κόκκινο χρώμα ως προεπιλογή. Αν η **useTexture** είναι 0, το **baseColor** τίθεται απευθείας στην τιμή του **fragmentColor**, παρακάμπτοντας την εφαρμογή υφής και χρησιμοποιώντας το προκαθορισμένο χρώμα του fragment.

Ακόμη, επεξεργαστήκαμε κατάλληλα τον **P1BVertexShader.vertexshader**

```
#version 330 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec4 vertexColor;
layout(location = 2) in vec2 vertexUV;

out vec4 fragmentColor;
out vec2 UV;

uniform mat4 MVP;

void main() {
   gl_Position = MVP * vec4(vertexPosition_modelspace, 1);
   fragmentColor = vertexColor;
   UV = vertexUV;
}
```

ώστε να επεξεργάζεται τα δεδομένα των κορυφών. Οι κορυφές του αντικειμένου, μαζί με τα χρώματα και τις UV συντεταγμένες, εισάγονται ως μεταβλητές εισόδου. Ο υπολογισμός της θέσης της κορυφής πραγματοποιείται μέσω του μετασχηματισμού **MVP** (Model-View-Projection Matrix), που εφαρμόζεται στην τρέχουσα θέση της κορυφής. Το χρώμα της κορυφής μεταβιβάζεται στο fragment shader μέσω της μεταβλητής εξόδου **fragmentColor**, ενώ οι UV συντεταγμένες αποστέλλονται με τη μεταβλητή **UV** για την εφαρμογή υφής. Ο shader εκτελεί τη βασική λειτουργία μετασχηματισμού και εξαγωγής δεδομένων από τις κορυφές, προετοιμάζοντας τα για το επόμενο στάδιο της επεξεργασίας.

2.2 Υλοποίηση του ερωτήματος (iv)

Στο παρόν ερώτημα, μας ζητήθηκε να υλοποιήσουμε τη δυνατότητα αλληλεπίδρασης μεταξύ του χαρακτήρα Α και του θησαυρού Β, όπως περιγράφεται στην εκφώνηση. Ο χρήστης καλείται να κινήσει τον χαρακτήρα Α ώστε να προλάβει να «ακουμπήσει» τον θησαυρό πριν αυτός αλλάξει θέση. Όταν ο χαρακτήρας ακουμπήσει τον θησαυρό, αυτός συρρικνώνεται στο μισό του αρχικού του μεγέθους και στη συνέχεια εξαφανίζεται. Η υλοποίηση έγινε με τρόπο που επιτρέπει στον χρήστη να παρατηρήσει τη διαδικασία της συρρίκνωσης πριν την εξαφάνιση του θησαυρού. Πιο συγκεκριμένα, έγιναν οι εξής αλλαγές:

Στο αρχείο **Source-1C.cpp**, υλοποιήσαμε την εντολή

```
void checkTreasureCollision(float* characterX, float* characterY, float* characterZ, float* treasureX, float* treasureY, float* treasureZ, GLfloat character_verticesB[], GLuint vertexbuffer3, int maze[10][10],, GLuint glowIntensityLoc) {

    static double shrinkStartTime = 0.0;

    if (!isTreasureVisible) return;

    if (fabs(*characterX - *treasureX) < 0.5f && fabs(*characterY - *treasureY) < 0.5f) {

        if (!isTreasureShrinking) {

            isTreasureShrinking = true;

            shrinkStartTime = glfwGetTime();

            glUniform1f(glowIntensityLoc, 1.0f);

        }

    }

    if (isTreasureShrinking) {

        double currentTime = glfwGetTime();

        float elapsedTime = static_cast<float>(currentTime - shrinkStartTime);

        if (elapsedTime < 1.0f) {

            float scaleFactor = 0.5f;

            GLfloat tempVertices[] = {

                *treasureX,*treasureY,*treasureZ,*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX,*treasureY,*treasureZ,

                *treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX,*treasureY,*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY - (0.8f * scaleFactor),

                *treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY,*treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ - (0.8f * scaleFactor),

                *treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX + (0.8f * scaleFactor),

                *treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),

                *treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY,*treasureZ,*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY,

                *treasureZ,*treasureX,*treasureY,*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY,*treasureZ,*treasureX,*treasureY,

                *treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY,*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY,*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY,

                *treasureZ - (0.8f * scaleFactor),*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX,*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor),*treasureX,
```

```
            *treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ,*treasureX + (0.8f * scaleFactor),*treasureY - (0.8f * scaleFactor),*treasureZ - (0.8f * scaleFactor)
        };
        std::copy(std::begin(tempVertices), std::end(tempVertices), character_verticesB);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3);
        glBufferData(GL_ARRAY_BUFFER, sizeof(tempVertices), tempVertices, GL_STATIC_DRAW);
      } else {
        isTreasureShrinking = false;
        isTreasureVisible = false;
        placeTreasure(maze, treasureX, treasureY, treasureZ, character_verticesB, characterX, characterY, vertexbuffer3, g_uv_buffer_data);
        GLfloat tempVertices[] = {
          *treasureX,*treasureY,*treasureZ,*treasureX,*treasureY - 0.8f,*treasureZ,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ,*treasureX,*treasureY,*treasureZ,
          *treasureX + 0.8f,*treasureY,*treasureZ,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ,*treasureX,*treasureY,*treasureZ - 0.8f,*treasureX,*treasureY - 0.8f,
          *treasureZ - 0.8f,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX,*treasureY,*treasureZ - 0.8f,*treasureX + 0.8f,*treasureY,*treasureZ - 0.8f,
          *treasureX + 0.8f,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX + 0.8f,*treasureY,*treasureZ,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ,*treasureX + 0.8f,
          *treasureY - 0.8f,*treasureZ - 0.8f,*treasureX + 0.8f,*treasureY,*treasureZ,*treasureX + 0.8f,*treasureY,*treasureZ - 0.8f,*treasureX + 0.8f,*treasureY,
          *treasureZ - 0.8f,*treasureX,*treasureY,*treasureZ,*treasureX,*treasureY - 0.8f,*treasureZ,*treasureX,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX,*treasureY,
          *treasureZ,*treasureX,*treasureY,*treasureZ - 0.8f,*treasureX,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX,*treasureY,*treasureZ,*treasureX,*treasureY,
          *treasureZ - 0.8f,*treasureX + 0.8f,*treasureY,*treasureZ - 0.8f,*treasureX,*treasureY,*treasureZ,*treasureX + 0.8f,*treasureY,*treasureZ,*treasureX + 0.8f,*treasureY,
          *treasureZ - 0.8f,*treasureX,*treasureY - 0.8f,*treasureZ,*treasureX,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ - 0.8f,*treasureX,
          *treasureY - 0.8f,*treasureZ,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ,*treasureX + 0.8f,*treasureY - 0.8f,*treasureZ - 0.8f
        };
        std::copy(std::begin(tempVertices), std::end(tempVertices), character_verticesB);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer3);
        glBufferData(GL_ARRAY_BUFFER, sizeof(tempVertices), tempVertices, GL_STATIC_DRAW);
        isTreasureVisible = true;
      }
    }
  }
```

σύμφωνα με την οποία, γίνεται ο χειρισμός της σύγκρουσης του χαρακτήρα Α με τον θησαυρό Β, της συρρίκνωσης του θησαυρού και της ανανέωσης της θέσης του. Αρχικά, ελέγχει αν ο θησαυρός είναι ορατός. Αν υπάρχει σύγκρουση, ενεργοποιεί τη διαδικασία συρρίκνωσης καταγράφοντας τον χρόνο έναρξης και αυξάνοντας τη φωτεινότητα. Κατά τη διάρκεια της συρρίκνωσης, υπολογίζει το ποσοστό μείωσης βάσει του χρόνου και προσαρμόζει τις συντεταγμένες των κορυφών του θησαυρού, ενημερώνοντας τον αντίστοιχο **buffer**. Αν η συρρίκνωση ολοκληρωθεί, ο θησαυρός εξαφανίζεται, τοποθετείται σε νέα θέση με το αρχικό του μέγεθος, και ο **buffer** ενημερώνεται με τις νέες συντεταγμένες. Με την ολοκλήρωση, η διαδικασία επανεκκινεί για τον επόμενο θησαυρό.

Τέλος, η κλήση στην **main**, έχει γίνει ως εξής:

```
checkTreasureCollision(&x, &y, &z, &treasureX, &treasureY, &treasureZ, charachter_verticesB, vertexbuffer3, maze, glowIntensityLoc);
```

2.3 Υλοποίηση του ερωτήματος (v)

υλοποιήθηκε με τον ίδιο τρόπο όπως στην Προγραμματιστική Άσκηση 1-Β

Ωστόσο, προστέθηκαν και οι λειτουργίες μετακίνησης της κάμερας στον άξονα x (**panning**) με τα πλήκτρα **<g>** και **<h>**, καθώς και στον άξονα y (**panning**) με τα πλήκτρα **<t>** και **<b>**. Πιο συγκεκριμένα, έγιναν οι εξής αλλαγές:

```
if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS) {
    if (!panXLeftKeyPressed) {
        std::cout << "Pan left key pressed\n";
        panOffsetX -= 0.1f;
        panXLeftKeyPressed = true;
    }
} else {
    panXLeftKeyPressed = false;
}
if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS) {
    if (!panXRightKeyPressed) {
        std::cout << "Pan right key pressed\n";
        panOffsetX += 0.1f;
        panXRightKeyPressed = true;
    }
} else {
    panXRightKeyPressed = false;
}
if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS) {
    if (!panYUpKeyPressed) {
        std::cout << "Pan up key pressed\n";
        panOffsetY += 0.1f;
        panYUpKeyPressed = true;
    }
} else {
    panYUpKeyPressed = false;
}
if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS) {
    if (!panYDownKeyPressed) {
        std::cout << "Pan down key pressed\n";
        panOffsetY -= 0.1f;
        panYDownKeyPressed = true;
    }
} else {
    panYDownKeyPressed = false;
}
```

```
cameraX = cameraDistance * cos(pitch) * sin(yaw);
cameraY = cameraDistance * sin(pitch);
cameraZ = cameraDistance * cos(pitch) * cos(yaw);
glm::vec3 cameraPos = glm::vec3(cameraX, cameraY, cameraZ);
glm::vec3 target = glm::vec3(panOffsetX, panOffsetY, 0.25f);
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
ViewMatrix = glm::lookAt(cameraPos, target, up);
```

Ο παραπάνω κώδικας χειρίζεται τη λειτουργία του panning για την κάμερα σε δύο άξονες (**X** και **Y**), επιτρέποντας την μετακίνηση της κάμερας προς τα αριστερά, δεξιά, πάνω ή κάτω με τα πλήκτρα **G**, **H**, **T**, και **B** αντίστοιχα. Για κάθε πλήκτρο, ο κώδικας ελέγχει αν είναι πατημένο και αν δεν έχει ήδη καταγραφεί το πάτημα του πλήκτρου, τότε τροποποιεί τις μετατοπίσεις **panOffsetX** ή **panOffsetY** και ενημερώνει την αντίστοιχη σημαία. Στη συνέχεια, υπολογίζονται οι συντεταγμένες της κάμερας (**cameraX**, **cameraY**, **cameraZ**) χρησιμοποιώντας τις παραμέτρους **pitch** και **yaw**. Η τελική θέση της κάμερας (**cameraPos**) συνδυάζεται με την ενημερωμένη θέση στόχευσης (**target**), η οποία περιλαμβάνει τις μετατοπίσεις του **panning**, και δημιουργείται το **view matrix** με τη χρήση της συνάρτησης **glm::lookAt**, η οποία καθορίζει την όψη της κάμερας σε σχέση με το σημείο στόχευσης και το καθορισμένο **up vector**.

2.4 Υλοποίηση των bonus ερωτημάτων

Για το ερώτημα (α):

η παρούσα υλοποίηση περιλαμβάνεται στο ερώτημα 2.1

Για το ερώτημα (β):

Το ερώτημα (β) δεν έχει υλοποιηθεί πλήρως, ωστόσο καταβλήθηκε προσπάθεια για την υλοποίηση του ειδικού εφέ με λάμψη. Πιο συγκεκριμένα, έγιναν οι εξής αλλαγές:

Στον **P1BFragmentShader.fragmentshader**, υλοποιήσαμε το εξής

```
uniform float glowIntensity;


  // inside main
  vec3 glowEffect = glowIntensity * vec3(0.0, 1.0, 0.0); // purple
  color = vec4(baseColor.rgb + glowEffect, baseColor.a);
```

Ο παραπάνω κώδικας υλοποιεί ένα εφέ λάμψης (**glow effect**) στο **fragment shader**. Η μεταβλητή **glowIntensity** ελέγχει την ένταση του εφέ, και χρησιμοποιείται για να υπολογιστεί η ένταση της λάμψης σε κάθε χρωματική συνιστώσα. Δημιουργείται ένας νέος διανύσματος **glowEffect**, ο οποίος προκύπτει από την ένταση της λάμψης και τη χρωματική σύνθεση που έχει επιλεγεί (σε αυτή την περίπτωση πράσινο, αντιπροσωπευόμενο από το διάνυσμα **vec3(0.0, 1.0, 0.0)**). Στη συνέχεια, το τελικό χρώμα του pixel υπολογίζεται προσθέτοντας το **glowEffect** στο βασικό χρώμα **baseColor.rgb** (που αντιστοιχεί στην RGB συνιστώσα του χρώματος του αντικειμένου), ενώ η αδιαφάνεια παραμένει αμετάβλητη με τη χρήση της **baseColor.a**. Έτσι, το εφέ λάμψης ενσωματώνεται στο χρώμα του αντικειμένου, προσδίδοντας του φωτεινότητα γύρω από τις επιλεγμένες περιοχές.

Στην checkTreasureCollision, σε περίπτωση που ταυτίζονται οι συντεταγμένες του χαρακτήρα A με τον θησαυρό B και αρχίζει να συρρικνώνεται, ενεργοποιείται η λάμψη

```
glUniform1f(glowIntensityLoc, 1.0f);
```

Στην **main**, έχω τον κώδικα

```
GLuint glowIntensityLoc = glGetUniformLocation(programID, "glowIntensity");
GLuint LightID = glGetUniformLocation(programID, "LightPosition_worldspace");

glm::vec3 lights = glm::vec3(4.0f, 4.0f, 4.0f);
glUniform3f(LightID, lightPos.x, lightPos.y, lightPos.z);
glUniform1f(glowIntensityLoc, 0.0f);

if (isTreasureShrinking) {
    double currentTime = glfwGetTime();
    float elapsedTime = static_cast<float>(currentTime - shrinkStartTime);

    float glowValue = glm::clamp(2.0f - elapsedTime, 0.0f, 2.0f);
    glUniform1f(glowIntensityLoc, glowValue);
} else {
    glUniform1f(glowIntensityLoc, 0.0f);
}
```

ο οποίος ρυθμίζει τη θέση του φωτός και την ένταση του εφέ λάμψης (**glow effect**) στο **fragment shader**. Αρχικά, η θέση του φωτός αποθηκεύεται στη μεταβλητή **lightPos** (που αντιπροσωπεύει μια τρισδιάστατη συντεταγμένη στον κόσμο) και στέλνεται στο shader μέσω του uniform **LightID** με την κλήση **glUniform3f**. Επιπλέον, η ένταση της λάμψης αρχικοποιείται με την τιμή **0.0f** μέσω του uniform **glowIntensityLoc**. Όταν η κατάσταση του θησαυρού είναι σε συρρίκνωση (όταν η μεταβλητή **isTreasureShrinking** είναι true), υπολογίζεται ο χρόνος που έχει περάσει από την έναρξη της συρρίκνωσης, και η ένταση της λάμψης υπολογίζεται με βάση αυτόν τον χρόνο. Η τιμή της λάμψης περιορίζεται με τη χρήση της συνάρτησης **glm::clamp** ώστε να παραμένει στο διάστημα [0.0, 2.0]. Αν δεν υπάρχει συρρίκνωση, η ένταση της λάμψης επανέρχεται στο **0.0f**. Έτσι, το φαινόμενο της λάμψης ελέγχεται δυναμικά ανάλογα με τη διάρκεια της συρρίκνωσης του θησαυρού.

Για το ερώτημα (γ):

Η προσθήκη φωτισμού μοντέλου **Phong** στον **fragment shader** περιλαμβάνει την υλοποίηση του φωτισμού που βασίζεται στις τρεις βασικές συνιστώσες: την περιβαλλοντική, τη διάχυτη και τη **specular** συνιστώσα. Για τη φωτεινή σημειακή πηγή, ορίζεται η θέση της στο (**10.0, 8.0, 4.0**) στον κόσμο και υπολογίζονται οι φωτιστικές συνιστώσες ανάλογα με τη θέση και την κατεύθυνση του φωτός σε σχέση με την επιφάνεια του αντικειμένου. Το περιβαλλοντικό φως προστίθεται ως μια βασική φωτεινότητα σε όλη την επιφάνεια, η διάχυτη συνιστώσα εξαρτάται από την γωνία μεταξύ του κατεύθυνσης του φωτός και της κατεύθυνσης του κανονικού της επιφάνειας, ενώ η **specular** συνιστώσα καθορίζεται από τη γωνία ανάκλασης του φωτός και τη θέση του θεατή, προσφέροντας την χαρακτηριστική λάμψη του αντικειμένου. Ο φωτισμός συνδυάζεται με τα δεδομένα χρώματος του αντικειμένου για την τελική απόδοση, δημιουργώντας μια ρεαλιστική αναπαράσταση φωτισμού σε **3D** περιβάλλον.

Στον **P1BVertexShader.vertexshader** έγιναν οι αλλαγές

```
uniform mat4 model;


out vec3 fragPosition;
out vec3 normal;
   fragPosition = vec3(model * vec4(vertexPosition_modelspace, 1.0));
   normal = mat3(transpose(inverse(model))) * vertexNormal;
```

οι οποίες υπολογίζουν τη θέση και τον κανονικό κάθε κορυφής στο παγκόσμιο σύστημα συντεταγμένων στον vertex shader. Η **fragPosition** υπολογίζεται με την εφαρμογή του **model matrix** στις κορυφές του αντικειμένου, μετατρέποντας τις από το τοπικό στο παγκόσμιο σύστημα. Ο κανονικός **normal** υπολογίζεται με τη χρήση του αντίστροφου και του μεταθετούμενου **model matrix** (μέσω της συνάρτησης **mat3(transpose(inverse(model)))**) για να εξασφαλιστεί ότι οι κανονικοί παραμένουν αμετάβλητοι κατά τη διάρκεια του μετασχηματισμού. Αυτές οι πληροφορίες στη συνέχεια μεταφέρονται στο fragment shader για την εφαρμογή φωτισμού.

Στον **P1BFragmentShader.fragmentshader** έγιναν οι αλλαγές

```
in vec3 fragPosition;
in vec3 normal;

uniform vec3 lightPosition;
uniform vec3 lightColor;
uniform vec3 viewPosition;
uniform vec3 objectColor;

vec3 norm = normalize(normal);
vec3 lightDir = normalize(lightPosition - fragPosition);
vec3 viewDir = normalize(viewPosition - fragPosition);
vec3 reflectDir = reflect(-lightDir, norm);

vec3 ambient = 0.2 * lightColor;

float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor * 1.5;
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
vec3 specular = spec * lightColor * 1.2;
vec4 phongColor = (ambient + diffuse + specular) * baseColor.rgb;
vec3 finalColor = phongColor + glowEffect;
color = vec4(finalColor, baseColor.a);
```

Ο παραπάνω κώδικας υλοποιεί τον φωτισμό μοντέλου Phong στο fragment shader. Ξεκινάει υπολογίζοντας τη κανονικοποίηση της κανονικής (**norm**) και τον υπολογισμό της κατεύθυνσης του φωτός (**lightDir**) και της κατεύθυνσης της θέας (**viewDir**) από την τρέχουσα θέση του fragment προς την πηγή φωτός και την κάμερα, αντίστοιχα. Επιπλέον, υπολογίζεται η κατεύθυνση της ανάκλασης του φωτός (**reflectDir**) χρησιμοποιώντας τη συνάρτηση **reflect**. Στη συνέχεια, ο φωτισμός Phong υπολογίζεται με τρεις συνιστώσες: **ambient** (περιβαλλοντικό φως), **diffuse** (διάχυτο φως), και **specular** (λάμψη), όπου κάθε συνιστώσα ενισχύεται με τον αντίστοιχο παράγοντα για να επιτευχθεί πιο έντονος φωτισμός. Η τελική τιμή του χρώματος του fragment προκύπτει από το συνδυασμό αυτών των φωτιστικών συνιστωσών και του χρώματος της υφής, προσθέτοντας και το glow effect. Το τελικό χρώμα του αντικειμένου υπολογίζεται ως το άθροισμα του φωτισμού Phong και του glow effect, το οποίο συνδυάζεται με την αδιαφάνεια της υφής.

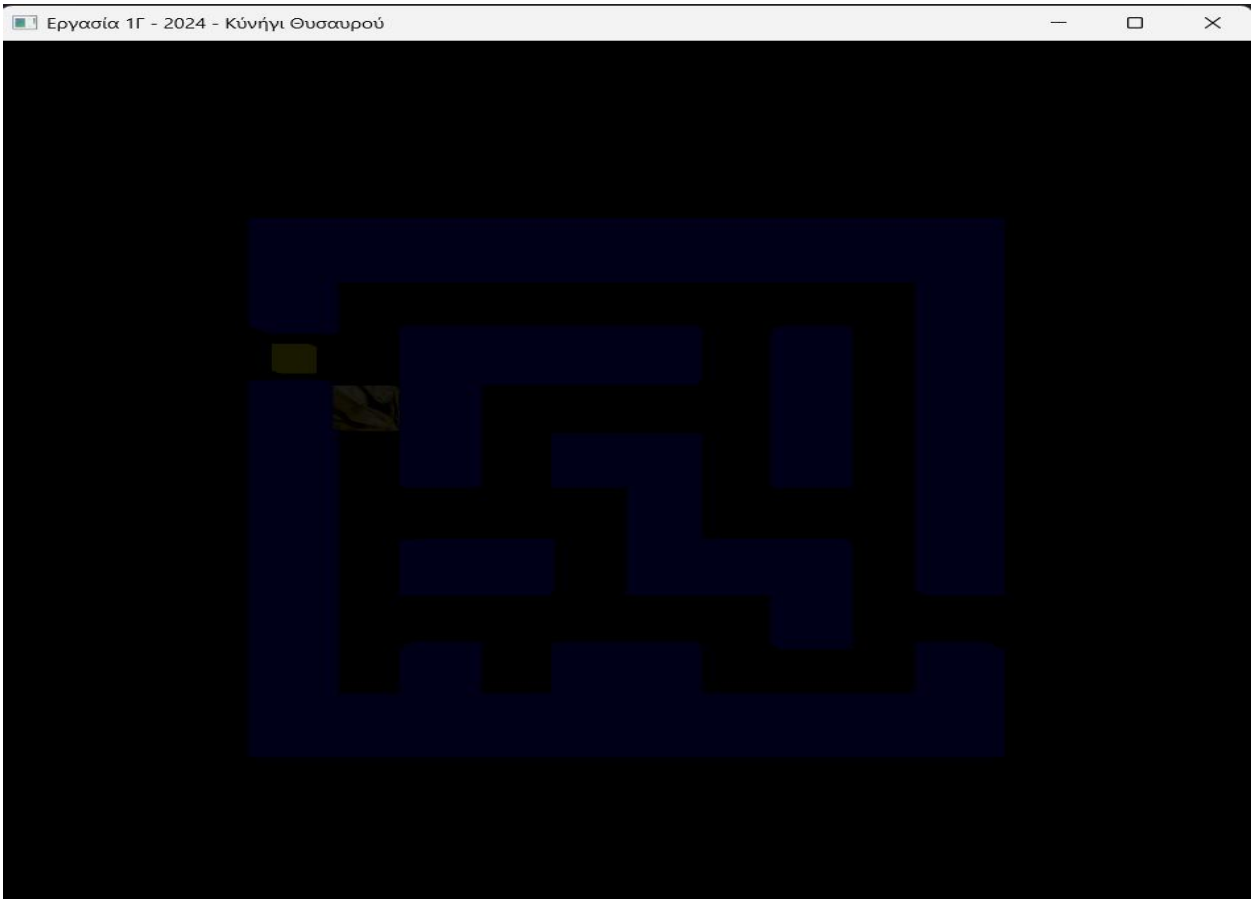
Στην main, υλοποιήθηκε ο κώδικας

```
glUseProgram(programID);
```

```
GLuint lightPosLoc = glGetUniformLocation(programID, "lightPosition");
GLuint lightColorLoc = glGetUniformLocation(programID, "lightColor");
GLuint viewPosLoc = glGetUniformLocation(programID, "viewPosition");
GLuint objectColorLoc = glGetUniformLocation(programID, "objectColor");


glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f);
glUniform3f(lightPosLoc, 10.0f, 8.0f, 4.0f);
glUniform3f(viewPosLoc, cameraX, cameraY, cameraZ);
glUniform3f(objectColorLoc, 1.0f, 1.0f, 1.0f);
```

Ο κώδικας ρυθμίζει τα **uniforms** για τον φωτισμό και τις παραμέτρους του αντικειμένου στο shader πρόγραμμα. Χρησιμοποιεί τη **glUseProgram** για να ενεργοποιήσει το πρόγραμμα και την **glGetUniformLocation** για να λάβει τις θέσεις των uniforms. Στη συνέχεια, αποδίδει τιμές στα uniforms με την **glUniform3f**, όπως το λευκό φως, τη θέση του φωτός, τη θέση της κάμερας και το χρώμα του αντικειμένου. Αυτό επιτρέπει στο shader να υπολογίσει σωστά τον φωτισμό και την εμφάνιση του αντικειμένου. Ωστόσο, το αποτέλεσμα είναι το εξής:



<u>Για το ερώτημα (δ):</u>

δεν υλοποιήθηκε

Αναφορές

- Ενδεικτικά video από την ιστοσελίδα του μαθήματος στο e-course (Βασιλική Σταμάτη).
- Rand and SRand