Πρώτο Μέρος (1Β) του Συνόλου Προγραμματιστικών Ασκήσεων
Γραφικά Υπολογιστών και Συστήματα Αλληλεπίδρασης

Ελευθέριος Ιωσηφίδης, 5233
Δανάη Χανλαρίδου, 5386

11/11/2025

1. Περιγραφή της εργασίας

(i) Για την επίλυση αυτού του ερωτήματος έχουμε το παρακάτω screenshot όπου φαίνεται η δημιουργία ενός 850*850 παραθύρου, με γκρι φόντο και ελληνικό τίτλο. Για την σωστή χρήση των unicodes, χρησιμοποιήσαμε το ChatGPT.

```
//(i) Window 850x850, dark gray bg, Greek title
if (!glfwInit()) { fprintf(stderr, "Failed to initialize GLFW\n"); return -1; }
glfwWindowHint(GLFW_SAMPLES, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

window = glfwCreateWindow(850, 850, u8"\u0395\u03C1\u03B3\u03B1\u03C3\u03AF\u03B1 1\u0392 \u2013 2025 \u2013 \u03A3\u03BA\u03B7\u03BD\u03B

if (!window) { fprintf(stderr, "Failed to open GLFW window\n"); glfwTerminate(); return -1; }
glfwMakeContextCurrent(window);
glewExperimental = true;
if (glewInit() != GLEW_OK) { fprintf(stderr, "Failed to initialize GLEW\n"); glfwTerminate(); return -1; }

glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
glClearColor(0.15f, 0.15f, 0.15f, 1.0f);        // dark grey
glEnable(GL_DEPTH_TEST);
```

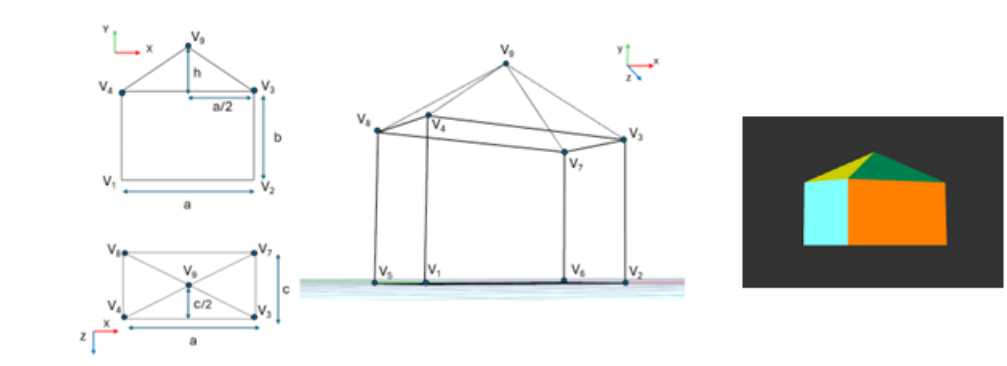
Τερματισμός με «1» υλοποιείται εδώ:

```
// quit with '1'
if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) glfwSetWindowShouldClose(window, GLFW_TRUE);

// move J/L one step per press
bool L = glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS;
```

(ii) Απαίτηση: Σχεδιασμός χαρακτήρα A (παραλληλεπίπεδο + πυραμίδα) & αποθήκευση τριγώνων
Έχουμε τις διαστάσεις από την εκφώνηση:

- a = 3, b = a/2 = 1.5, c = 2, h = a/4 = 0.75.
- Δίνεται apex (κορυφή): v9 = (0, −7.75, 1.0).
- Πάνω επιφάνεια βάσης: yTop = v9.y − h = −7.75 − 0.75 = −8.5
- Κάτω επιφάνεια βάσης: yBot = yTop − b = −8.5 − 1.5 = −10.0
- x-όρια: xL = −a/2 = −1.5, xR = +a/2 = +1.5
- z-όρια: zF = 0 (μπροστά), zB = c = 2 (πίσω)

  Το σχεδιαστικό κομμάτι, σύμφωνα με το οποίο υπολογίσαμε όλες τις συντεταγμένες, όπως δίνεται στην εκφώνηση είναι:



Εικόνα 2 – (πάνω αριστερά) Η μπροστινή όψη του A (front view), (κάτω αριστερά) η όψη του A από πάνω (top view), (μέση) η προβολή του A υπό γωνία, (δεξιά) ο χρωματισμός του A.

Κορυφές παραλληλεπιπέδου (v1…v8), με όρο «v1..v4 πάνω στο z=0»:

- v1(xL, yBot, zF) = (−1.5, −10.0, 0)
- v2(xR, yBot, zF) = (+1.5, −10.0, 0)
- v3(xR, yTop, zF) = (+1.5, −8.5, 0)
- v4(xL, yTop, zF) = (−1.5, −8.5, 0)
- v5(xL, yBot, zB) = (−1.5, −10.0, -2)
- v6(xR, yBot, zB) = (+1.5, −10.0, -2)
- v7(xR, yTop, zB) = (+1.5, −8.5, -2)
- v8(xL, yTop, zB) = (−1.5, −8.5, -2)
- v9 = (0, −7.75, 1)

Οι συντεταγμένες υλοποιούνται σε αυτό το σημείο του κώδικα:

```
// Build the character A vertices (base box + 4 roof tris) from the spec
static void build_A(vector<glm::vec3>& verts, vector<glm::vec4>& cols) {
    // apex v9 (your spec with back at z = -2 and apex z = -1)
    glm::vec3 v9(0.0f, -7.75f, -1.0f);

    float yTop = v9.y - A_roof; // -8.50
    float yBot = yTop - A_hgt;  // -10.00
    float xL = -A_len * 0.5f;   // -1.5
    float xR = A_len * 0.5f;    //  1.5
    float zF = 0.0f;           // front
    float zB = -A_dep;          // back = -2

    glm::vec3 v1(xL, yBot, zF), v2(xR, yBot, zF), v3(xR, yTop, zF), v4(xL, yTop, zF);
    glm::vec3 v5(xL, yBot, zB), v6(xR, yBot, zB), v7(xR, yTop, zB), v8(xL, yTop, zB);
```

Στο παρακάτω screenshot φαίνεται πως γεμίζονται vector<glm::vec3> Averts με κορυφές (triplets) και vector<glm::vec4> Acols με χρώματα ανά τρίγωνο.

```
//(ii)+ (iii) Character A geometry + per-face colors
vector<glm::vec3> Averts; vector<glm::vec4> Acols;
build_A(Averts, Acols);

GLuint vboA, cboA;
glGenBuffers(1, &vboA); glBindBuffer(GL_ARRAY_BUFFER, vboA);
glBufferData(GL_ARRAY_BUFFER, Averts.size() * sizeof(glm::vec3), Averts.data(), GL_STATIC_DRAW);
glGenBuffers(1, &cboA); glBindBuffer(GL_ARRAY_BUFFER, cboA);
glBufferData(GL_ARRAY_BUFFER, Acols.size() * sizeof(glm::vec4), Acols.data(), GL_STATIC_DRAW);
```

Η αποστολή στη GPU γίνεται ως εξής:

```
GLuint vboA, cboA;
glGenBuffers(1, &vboA); glBindBuffer(GL_ARRAY_BUFFER, vboA);
glBufferData(GL_ARRAY_BUFFER, Averts.size() * sizeof(glm::vec3), Averts.data(), GL_STATIC_DRAW);
```

(iii) Απαίτηση: Διαφορετικά χρώματα σε κάθε πλευρά του Α

- Σημεία κώδικα: πάλι στη συνάρτηση build_A(...) ορίζονται 10 διαφορετικά χρώματα (6 επιφάνειες βάσης + 4 της πυραμίδας) και με tint(cols, ...) αντιστοιχίζονται ανά τρίγωνο.
- Στο παρακάτω screenshot έχουμε, την πρώτη εντολή να σημαίνει "ενεργοποίησε το attribute 1 (τα χρώματα) ώστε να διαβαστούν από το buffer", η δεύτερη εντολή σημαίνει "Από εδώ και πέρα, οποιαδήποτε ρύθμιση attribute ή σχεδίαση, θα διαβάζει δεδομένα από το buffer cboA." και η τρίτη εντολή σημαίνει "Για το attribute 1 (χρώμα), διάβαζε 4 floats (R,G,B,A) συνεχόμενα για κάθε κορυφή, ξεκινώντας από το byte 0 του buffer.". Ομοίως για τις γραμμές 353-355.

```
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, cboA);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vboA);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

Εδώ, φαίνεται ο χρωματισμός του Α με διαφορετικές αποχρώσεις ανά πλευρά. Τα cFront, cBack, cLeft, cRight, cTop, cBot είναι οι 6 αποχρώσεις για τις πλευρές του βασικού παραλληλεπιπέδου ("βάσης"). Τα r1, r2, r3, r4 είναι οι 4 διαφορετικές αποχρώσεις για τα τρίγωνα της πυραμίδας ("στέγης") του Α. Κάθε μία είναι glm::vec4(R, G, B, A) με τιμές από 0–1.

```
// Base faces (each side a different color)
glm::vec4 cFront(1.0, 0.0, 0.0, 1.0);   // bright red
glm::vec4 cBack(0.0, 0.8, 0.0, 1.0);    // green
glm::vec4 cLeft(0.0, 0.4, 1.0, 1.0);    // blue-cyan
glm::vec4 cRight(1.0, 0.6, 0.0, 1.0);   // orange
glm::vec4 cTop(0.8, 0.0, 0.8, 1.0);     // violet
glm::vec4 cBot(0.4, 0.4, 0.4, 1.0);     // gray
// Roof triangles (four distinct colors)
glm::vec4 r1(0.0, 1.0, 1.0, 1.0);       // cyan
glm::vec4 r2(1.0, 0.0, 1.0, 1.0);       // magenta
glm::vec4 r3(1.0, 1.0, 0.0, 1.0);       // yellow
glm::vec4 r4(0.6, 0.2, 1.0, 1.0);       // purple-blue
```

Ακριβώς παρακάτω, σε κάθε επιφάνεια του A, καλείται:

```
// Base (12 tris)
// front z=0
tri(verts, v1, v2, v3); tint(cols, cFront);
tri(verts, v1, v3, v4); tint(cols, cFront);
// back z=-2
tri(verts, v5, v6, v7); tint(cols, cBack);
tri(verts, v5, v7, v8); tint(cols, cBack);
// left x=xL
tri(verts, v1, v4, v8); tint(cols, cLeft);
tri(verts, v1, v8, v5); tint(cols, cLeft);
// right x=xR
tri(verts, v2, v6, v7); tint(cols, cRight);
tri(verts, v2, v7, v3); tint(cols, cRight);
// top y=yTop
tri(verts, v4, v3, v7); tint(cols, cTop);
tri(verts, v4, v7, v8); tint(cols, cTop);
// bottom y=yBot
tri(verts, v1, v5, v6); tint(cols, cBot);
tri(verts, v1, v6, v2); tint(cols, cBot);

// Roof (4 tris) with apex v9
tri(verts, v4, v3, v9); tint(cols, r1);
tri(verts, v3, v7, v9); tint(cols, r2);
tri(verts, v7, v8, v9); tint(cols, r3);
tri(verts, v8, v4, v9); tint(cols, r4);
```

Η tri() παίρνει τρεις κορυφές ενός τριγώνου (A, B, C) και τις προσθέτει (append) στο vector v, το οποίο περιέχει όλες τις κορυφές του αντικειμένου. Η tri(...) προσθέτει τις 3 κορυφές (γεωμετρία), ενώ η tint(...) προσθέτει το ίδιο χρώμα για τις 3 κορυφές. Για tri(verts, v1, v2, v3) σημαίνει = πρόσθεσε στο vector verts τα σημεία v1, v2, v3 με αυτή τη σειρά.

```
static void tri(vector<glm::vec3>& v, glm::vec3 A, glm::vec3 B, glm::vec3 C) {
    v.push_back(A); v.push_back(B); v.push_back(C);
}
static void tint(vector<glm::vec4>& c, glm::vec4 col) {
    c.push_back(col); c.push_back(col); c.push_back(col);
}
```

(iv) Απαίτηση: Αρχική κάμερα (0, −5, 20), στόχος (0,0,0), up (0,1,0), FOV=60° (σταθερό)

Μέσα στην συνάρτηση camera_function ορίζουμε τα εξής:

```
// --- Κατάσταση κάμερας (orbit) ---
static float r = glm::length(glm::vec3(0.0f, -5.0f, 20.0f));
static float pitch = std::asin(-5.0f / r);
static float yaw   = 0.0f;
```

Ο προσανατολισμός/προβολή δίνεται από:

```
static const float aspect = 1.0f; // 850/850
if (!init) {
    ProjectionMatrix = glm::perspective(glm::radians(FOVdeg), aspect, 0.1f, 200.0f);
    init = true;
}
```

Στη γραμμή 102 (ViewMatrix = glm::lookAt(eye, glm::vec3(0,0,0), glm::normalize(up));) το glm::lookAt κοιτάζει στο (0,0,0) με up ~(0,1,0) (προκύπτει από τον quaternion προσανατολισμό, αρχικά είναι το κλασικό world-up).

Από StackOverflow βρήκαμε ότι ένα Quaternion είναι ένα 3D διάνυσμα με περιστροφή (vector with a rotation). https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/

```
glm::vec3 eye = camPos();
glm::mat4 View = glm::lookAt(eye, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));
```

Σημαντικό είναι να θυμόμαστε ότι FOV=60° ορίζεται μία φορά και δεν αλλάζει στη διάρκεια εκτέλεσης. Η αρχική θέση εξάγεται από (r, pitch, yaw) έτσι ώστε eye ≈ (0,−5,20).

(v) Απαίτηση: Κίνηση A στον άξονα x, βήμα a/2 με L/J

Παρακάτω φαίνεται ο έλεγχος πλήκτρων (μία «θέση» ανά πάτημα). Δεν υπάρχει όριο στην κίνηση του A, δηλαδή μπορεί να βγει και εκτός παραθύρου.

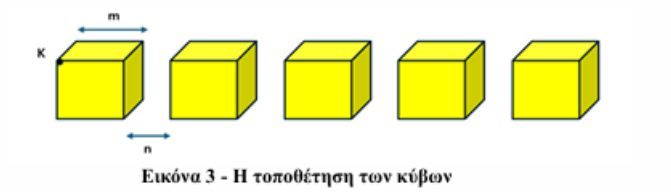
```
// (v) A moves by a/2 on X with J/L (edge-trigger)
float AoffsetX = 0.0f; const float step = A_len * 0.5f;     // 1.5
bool pL = false, pJ = false, pW = false, pX = false, pQ = false, pZ = false, pPLUS = false, pMINUS = false;

while (!glfwWindowShouldClose(window))
{
    // quit with '1'
    if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) glfwSetWindowShouldClose(window, GLFW_TRUE);

    // move J/L one step per press
    bool L = glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS;
    bool J = glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS;
    if (L && !pL) AoffsetX += step;
    if (J && !pJ) AoffsetX -= step;
    pL = L; pJ = J;
```

Παρακάτω φαίνεται η εφαρμογή μετατόπισης στο μοντέλο του A.

```
// draw A
glm::mat4 ModelA = glm::translate(glm::mat4(1.0f), glm::vec3(AoffsetX, 0, 0));
glm::mat4 MVP_A = P * V * ModelA;
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP_A[0][0]);
```

Σημαντικό σε αυτό το ερώτημα είναι ότι χρησιμοποιούμε edge-trigger λογική με δύο flags (pL, pJ) για να γίνεται ακριβώς ένα βήμα ανά πάτημα. Το βήμα είναι a/2 όπως ζητείται.

(vi) Απαίτηση: Σειρά από 5 κύβους επάνω από τον A – K = (−9, 10, 0), m=2, n=2. Η εικόνα δίνεται που δίνεται στην εκφώνηση για κατανόηση του σχεδιασμού των κύβων:



Εικόνα 3 - Η τοποθέτηση των κύβων

Στο παρακάτω screenshot δίνονται οι ορισμοί παραμέτρων & αρχικής κορυφής K και η δημιουργία 5 κύβων με σταθερό βήμα stride. Όλοι οι κύβοι είναι ίδιου μεγέθους (m=2) και απέχουν n=2, συνολικό stride=4 κατά x. Ο πρώτος κύβος έχει top-front-left K στο (−9, 10, 0) όπως ζητείται.

```
//(vi) 5 cubes above A
vector<glm::vec3> cubes;
const float m = 2.0f, n = 2.0f, stride = m + n; // 4.0
glm::vec3 K0(-9.0f, 10.0f, 0.0f);               // first cube K
for (int i = 0; i < 5; i++) build_cube(cubes, K0 + glm::vec3(stride * i, 0, 0), m);
```

Με την βοήθεια του ChatGPT προσθέσαμε την βοηθητική συνάρτηση build_cube(...) η οποία δημιουργεί 12 τρίγωνα (6 επιφάνειες × 2 τρίγωνα η καθεμία) και τα αποθηκεύει στο διάνυσμα out, ώστε ο κύβος να μπορεί να σχεδιαστεί με glDrawArrays(GL_TRIANGLES, …).

Πρώτη γραμμή: Αυτές είναι οι 4 κορυφές του μπροστινού τετραγώνου (front face), στο επίπεδο z = 0. Το -m στον y σημαίνει ότι το "κάτω" είναι προς τα αρνητικά y (οπότε το πάνω είναι το K).

Δευτέρη γραμμή: δίνονται οι 4 κορυφές (με συντεταγμένες) του πίσω τετραγώνου (back face) στο επίπεδο z = m.

Τρίτη γραμμή: Αυτή είναι η λάμδα συνάρτηση. Δημιουργεί ένα μικρό βοηθητικό "alias" T(...) που καλεί τη συνάρτηση tri(...). Η tri(...) απλώς προσθέτει τρεις κορυφές στο vector out. Έτσι, κάθε T(P, Q, R) ορίζει ένα τρίγωνο.

Υπόλοιπες γραμμές (267-272): Ο κύβος έχει 6 πλευρές. Κάθε πλευρά είναι τετράγωνο άρα 2 τρίγωνα. Κάθε T(...) προσθέτει 3 κορυφές →12 * 3 = 36 κορυφές συνολικά ανά κύβο.

```
// one cube of size m with top-front-left corner K
static void build_cube(vector<glm::vec3>& out, glm::vec3 K, float m) {
    glm::vec3 A = K, B = K + glm::vec3(m, 0, 0), C = K + glm::vec3(m, -m, 0), D = K + glm::vec3(0, -m, 0);
    glm::vec3 E = K + glm::vec3(0, 0, m), F = K + glm::vec3(m, 0, m), G = K + glm::vec3(m, -m, m), H = K + glm::vec3(0, -m, m);
    auto T = [&](glm::vec3 p, glm::vec3 q, glm::vec3 r) { tri(out, p, q, r); };
    T(A, B, C); T(A, C, D);         // front
    T(E, G, H); T(E, F, G);         // back
    T(A, D, H); T(A, H, E);         // left
    T(B, F, G); T(B, G, C);         // right
    T(A, E, F); T(A, F, B);         // top
    T(D, C, G); T(D, G, H);         // bottom
}
```

Άρα οι θέσεις τους (πάνω στον άξονα x) είναι:

(-9, 10, 0), (-5, 10, 0), (-1, 10, 0), (3, 10, 0), (7, 10, 0)

Ο σχεδιασμός των κύβων γίνεται παρακάτω. Συνδυάζεται ο πίνακας cubes με την πανοραμική κάμερα (Projection * View). Χρησιμοποιεί το View που ορίζεται λίγο πιο πάνω.

```
// draw cubes (single solid color via constant attrib)
glm::mat4 MVP_I = P * V * glm::mat4(1.0f);
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP_I[0][0]);

glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vboCubes);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glDisableVertexAttribArray(1);
glVertexAttrib4f(1, 1.0f, 1.0f, 0.0f, 1.0f); // yellow cubes

glDrawArrays(GL_TRIANGLES, 0, (GLsizei)cubes.size());
glDisableVertexAttribArray(0);
```

Χρώμα κύβων: έστω σταθερό κίτρινο:

```
glDisableVertexAttribArray(1);
glVertexAttrib4f(1, 1.0f, 1.0f, 0.0f, 1.0f);
```

(vii) Απαίτηση: Κάμερα με πληκτρολόγιο (press-only): W/X (άξονας x), Q/Z (άξονας y), Numpad +/− (zoom)

Στο παρακάτω screenshot (ομοίως με ερώτημα iv) φαίνεται το σταθερό FOV στην camera_function() (ορίζεται μία φορά, δεν αλλάζει):

```
if (!init) {
    ProjectionMatrix = glm::perspective(glm::radians(FOVdeg), aspect, 0.1f, 200.0f);
    init = true;
}
```

Ο χειρισμός πλήκτρων (μόνο έλεγχος για GLFW_PRESS) φαίνεται ότι υλοποιείται παρακάτω. Εδώ χρησιμοποιείται dt (διαφορά χρόνου) για ομαλή κίνηση ανά δευτερόλεπτο.

```
// --- Συνεχής έλεγχος πλήκτρων ---
if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS) yaw   += yawSpeed   * dt;
if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS) yaw   -= yawSpeed   * dt;
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) pitch += pitchSpeed * dt;
if (glfwGetKey(window, GLFW_KEY_X) == GLFW_PRESS) pitch -= pitchSpeed * dt;

if (glfwGetKey(window, GLFW_KEY_KP_ADD)      == GLFW_PRESS) r -= zoomSpeed * dt; // zoom in
if (glfwGetKey(window, GLFW_KEY_KP_SUBTRACT) == GLFW_PRESS) r += zoomSpeed * dt; // zoom out
r = glm::clamp(r, 2.0f, 80.0f);
```

Στο παρακάτω screenshot φαίνεται πως υλοποιούνται οι 360° περιστροφές (wrap σε 0..2π) και η αποφυγή gimbal lock με quaternions (δόθηκε παραπάνω ο ορισμός). Χρησιμοποιούνται quaternions (qYaw * qPitch) για σταθερό προσανατολισμό χωρίς κλειδώματα γωνιών διαφορετικά “κολλούσαν” στις 180 μοίρες.

```
// --- Wrap και στους δύο άξονες (πλήρης 360° επαναλαμβανόμενα) ---
const float TWO_PI = glm::two_pi<float>();
yaw   = std::fmod(yaw,   TWO_PI); if (yaw   < 0.0f) yaw   += TWO_PI;
pitch = std::fmod(pitch, TWO_PI); if (pitch < 0.0f) pitch += TWO_PI;

// --- Προσανατολισμός με quaternions (χωρίς gimbal lock) ---
// σειρά: yaw γύρω από world-Y και μετά pitch γύρω από local-X
glm::quat qYaw   = glm::angleAxis(yaw,   glm::vec3(0,1,0));
glm::quat qPitch = glm::angleAxis(pitch, glm::vec3(1,0,0));
glm::quat q = qYaw * qPitch;

// Από τον προσανατολισμό βγάζουμε eye & up
glm::vec3 eye = q * glm::vec3(0, 0, r);     // περιστρεφουμε το (0,0,r)
glm::vec3 up  = q * glm::vec3(0, 1, 0);     // το “πάνω” της κάμερας

ViewMatrix = glm::lookAt(eye, glm::vec3(0,0,0), glm::normalize(up));
```

Εξήγηση τι κανει η συνάρτηση camera_function():

Πιο αναλυτικά:

- Στην αρχή, η συνάρτηση δημιουργεί τον πίνακα προβολής (ProjectionMatrix) με σταθερό οπτικό πεδίο 60° (FOV) και λόγο πλευρών 1:1. Αυτή η ρύθμιση γίνεται μόνο μία φορά.
- Ορίζει την αρχική θέση της κάμερας με απόσταση r από το κέντρο, έτσι ώστε η κάμερα να βρίσκεται περίπου στο σημείο (0, −5, 20) και να κοιτάζει προς το (0,0,0). Οι γωνίες yaw και pitch αντιπροσωπεύουν τις περιστροφές γύρω από τους άξονες Y και X αντίστοιχα.
- Υπολογίζει τη χρονική διαφορά dt μεταξύ διαδοχικών καρέ, ώστε η κίνηση να είναι ομαλή και ανεξάρτητη από τον ρυθμό καρέ.
- Ανάλογα με τα πλήκτρα που πατά ο χρήστης:
    - Q / Z περιστρέφουν την κάμερα γύρω από τον άξονα Y (yaw),
    - W / X περιστρέφουν γύρω από τον άξονα X (pitch),
    - + / – του αριθμητικού πληκτρολογίου κάνουν zoom in/out μεταβάλλοντας την απόσταση r.
- Οι γωνίες yaw και pitch "τυλίγονται" περιοδικά (0–360°) ώστε η περιστροφή να μπορεί να συνεχίζεται απεριόριστα.
- Χρησιμοποιεί quaternions (glm::quat) για να υπολογίσει τον τελικό προσανατολισμό, αποφεύγοντας το φαινόμενο gimbal lock που συμβαίνει με τις απλές περιστροφές Euler.
- Τέλος, υπολογίζει τη νέα θέση της κάμερας (eye) και το διάνυσμα "πάνω" (up), και δημιουργεί τον πίνακα θέασης (ViewMatrix) μέσω της συνάρτησης glm::lookAt(), ώστε η κάμερα να κοιτάζει πάντα το κέντρο της σκηνής

Τέλος, για την υλοποίηση όλων των παραπάνω, προσθέσαμε της εξής βιβλιοθήκες οι οποίες επιτρέπουν ομαλές, συνεχείς και φυσικές περιστροφές της κάμερας στον 3D χώρο χωρίς παραμορφώσεις ή περιορισμούς.

```
#include <glm/gtc/constants.hpp> // για glm::two_pi<float>()
#include <glm/gtc/quaternion.hpp>
#include <glm/gtx/quaternion.hpp>
```

#include <glm/gtc/constants.hpp> = για μαθηματικές σταθερές (π, 2π, κ.λπ.)

Στον κώδικα χρησιμοποιείται η glm::two_pi<float>() για να επαναφέρει (wrap) τις γωνίες yaw και pitch όταν ξεπερνούν τις 360°, ώστε να συνεχίζουν κυκλικά χωρίς υπερχείλιση.

#include <glm/gtc/quaternion.hpp> = για βασικούς ορισμούς quaternions

Τα quaternions χρησιμοποιούνται για την αναπαράσταση περιστροφών στο 3D χώρο, αποφεύγοντας προβλήματα όπως το gimbal lock που εμφανίζεται όταν χρησιμοποιούνται διαδοχικές περιστροφές Euler. Στον κώδικα μας παριέχει την συνάρτηση:

- glm::angleAxis(angle, axis) → δημιουργεί quaternion περιστροφής γύρω από άξονα

#include <glm/gtx/quaternion.hpp> = για προχωρημένες πράξεις και εφαρμογές quaternions

Στον κώδικα, χρησιμοποιείται για να εφαρμόσει την περιστροφή της κάμερας στα διανύσματα θέσης και "πάνω" (eye, up) με την εντολή:

```
glm::vec3 eye = q * glm::vec3(0, 0, r);
glm::vec3 up  = q * glm::vec3(0, 1, 0);
```

όπου q είναι το quaternion που προκύπτει από συνδυασμό των περιστροφών yaw και pitch.

2. Πληροφορίες σχετικά με την υλοποίηση

Η εφαρμογή γράφτηκε σε C++ και χρησιμοποιήσαμε Windows 10 (64-bit). Η εφαρμογή τρέχει ως εκτελέσιμο console application σε περιβάλλον Windows, με χρήση GLFW, GLEW και OpenGL. Η ανάπτυξη του κώδικα έγινε σε Visual Studio. Ενεργοποιήσαμε το σωστό working directory ώστε τα shader αρχεία P1BVertexShader.vertexshader και P1BFragmentShader.fragmentshader να είναι προσβάσιμα στο runtime από τον φάκελο που τρέχει το .cpp. Σε περιβάλλον Windows, προτείνεται # define NOMINMAX πριν από τυχόν #include <windows.h>.

3. Αξιολόγηση

Χωρίσαμε την εργασία σε διακριτά μέρη:

<u>ΑΜ 5233:</u>

Υλοποίησε τη γεωμετρία του χαρακτήρα Α, με αναλυτικό υπολογισμό όλων των κορυφών v1–v9 και δημιουργία των αντίστοιχων τριγώνων της βάσης και της πυραμίδας. Δημιούργησε τη συνάρτηση build_A() με την οποία παράγονται οι πίνακες κορυφών (Averts) και χρωμάτων (Acols), καθώς και τις βοηθητικές συναρτήσεις tri() και tint(). Ανέλαβε τον χρωματισμό των πλευρών του χαρακτήρα Α. Ανέπτυξε τη συνάρτηση build_cube() για τη δημιουργία των κύβων της σκηνής και φρόντισε για την τοποθέτησή τους στη σωστή διάταξη (αρχική κορυφή K = (−9,10,0), m = 2, n = 2).

<u>ΑΜ 5386:</u>

Δημιούργησε την αρχικοποίηση του παραθύρου, του GLFW/GLEW περιβάλλοντος και τη διαμόρφωση του OpenGL context. Διαμόρφωσε το παράθυρο σύμφωνα με την εκφώνηση. Ανέπτυξε τη λογική του τερματισμού με πλήκτρο “1” και του κινητού χαρακτήρα Α πάνω στον άξονα x με πλήκτρα L/J, εισάγοντας edge-triggered λογική για ομαλή μετακίνηση. Υλοποίησε τη συνάρτηση camera_function(), η οποία διαχειρίζεται την περιστροφή και το zoom της κάμερας με τα πλήκτρα W, X, Q, Z, +, −, υπολογίζοντας σε κάθε καρέ τη νέα θέση του παρατηρητή σε σφαιρικές συντεταγμένες (yaw, pitch, radius).

<u>Μαζί:</u> Ενσωμάτωσαμε τον έλεγχο πληκτρολογίου (keyboard handling) με glfwGetKey()
ώστε να καλύπτονται όλες οι απαιτούμενες κινήσεις. Εξασφάλισαμε ότι το FOV παραμένει σταθερό στα 60°, ενώ το zoom γίνεται μέσω αλλαγής της ακτίνας παρατήρησης camR.

<u>Δυσκολίες:</u>

1) Χειρισμός press-only input και σταθερό FOV
2) Η κατανόηση περιστροφής της κάμερας

Η συνεργασία χαρακτηρίστηκε από καλή επικοινωνία, διαχωρισμό αρμοδιοτήτων και συνεχείς δοκιμές. Συνεργαστήκαμε τόσο εξ αποστάσεως (μέσω Git) όσο και δια ζώσης για δοκιμές και οπτική επαλήθευση του αποτελέσματος.

4. Αναφορές – Πηγές που χρησιμοποιήθηκαν κατά την εκπόνηση της εργασίας.

- OpenGL Tutorials (Matrices, MVP, VBO/VAO, Shaders) - https://learnopengl.com/Advanced-OpenGL/Geometry-Shader
- Για glm::perspective, glm::lookAt - https://learnopengl.com/Getting-started/Camera
- Για input handling και key codes - https://www.glfw.org/docs/3.0/group__input.html
- Youtube tutorials για κατανόηση της έννοιας "κάμερας" -

  https://www.youtube.com/watch?v=lsOoo3Q4_ag

  https://www.youtube.com/watch?v=U0_ONQQ5ZNM
- Σημειώσεις και διαφάνειες του μαθήματος
- ChatGPT για επίλυση απορίων, κατανόηση της λειτουργίας της κάμερας και της βελτιστοποίησης του κώδικα.