Τρίτο Μέρος (1Γ) του Συνόλου Προγραμματιστικών Ασκήσεων

Γραφικά Υπολογιστών και Συστήματα Αλληλεπίδρασης

Ελευθέριος Ιωσηφίδης, 5233

Δανάη Χανλαρίδου, 5386

28/11/2025

1. Περιγραφή της εργασίας

Σκοπός της εργασίας 1Γ είναι η δημιουργία ενός απλού 3D παιχνιδιού σε OpenGL 3.3, στο οποίο ο χαρακτήρας Α πυροβολεί κύβους-εχθρούς που κατεβαίνουν προς το μέρος του. Η υλοποίηση βασίζεται στην άσκηση 1Β και επεκτείνεται με:

- υφή στον χαρακτήρα Α,
- βλήματα / συγκρούσεις,
- κάμερα ελεγχόμενη από το πληκτρολόγιο,
- bonus λειτουργίες (οπτικό/ηχητικό εφέ, pause/restart, επιτάχυνση/επιβράδυνση εχθρών, οριζόντιο μοτίβο κίνησης, φωτισμός Phong και μετακίνηση φωτεινής πηγής).

(i) Απαιτήσεις - Παράθυρο, τίτλος, φόντο, τερματισμός

Ο κώδικας είναι ίδιος με αυτόν της άσκησης 1Β. Χρησιμοποιείται η GLFW για τη δημιουργία παραθύρου 850x850. Ο τίτλος είναι σε UTF-8 κωδικοποίηση την οποία πήραμε από το ChatGPT.

```
// (i) Παράθυρο 850x850, σκούρο γκρι, τίτλος 1Γ
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\u0393 – 2025 – \u039A\u03B1\u03C4\u03B1\u03C3\u03C4\u03C1\u03BF\u03C6\u03AD\u03B1\u03C2"
    NULL, NULL);
```

Το background ορίζεται σε σκούρο γκρι με:

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

Η εφαρμογή τερματίζεται όταν πατηθεί το πλήκτρο 1 μέσα στο main loop:

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

(ii) Απαιτήσεις- Χαρακτήρας Α και υφή

Ο κώδικας είναι αυτός που είχαμε και στην άσκηση 1Β με την επιπλέον προσθήκη της υφής για τον χαρακτήρα Α. Οι διαστάσεις του Α ορίζονται με σταθερές:

```
////////////////////////////////////////////////
// --- A dimensions (όπως στην 1B) ---
static const float A_len = 3.0f;          // a
static const float A_hgt = A_len * 0.5f;  // b = a/2 = 1.5
static const float A_dep = 2.0f;          // c = 2
static const float A_roof = A_len * 0.25f; // h = a/4 = 0.75
```

Η κορυφή v9 (άκρη της σκεπής) τοποθετείται στο

```
// θέση της v9 στο local space του A
static const glm::vec3 A_apex_local(0.0f, -7.75f, -1.0f);
```

Η συνάρτηση build_A (όπως και στην 1Β) υπολογίζει τις κορυφές v1..v8 της βάσης (ορθογώνιο παραλληλεπίπεδο) και προσθέτει τέσσερα τρίγωνα οροφής που ενώνουν v3, v4, v8, v7 με το v9. Τα τρίγωνα αποθηκεύονται σε std::vector<glm::vec3> Averts.

Υπολογισμός uv συντεταγμένων

- Για κάθε παραλληλόγραμμη πλευρά αντιστοιχίζουμε την υφή στο εύρος [0,1]x[0,1]. Παράδειγμα front πλευρά (z=0):

```
// Βασικά τετράπλευρα: απλό mapping 0..1
glm::vec2 t1(0, 0), t2(1, 0), t3(1, 1), t4(0, 1);

// UV για τις πλευρές (left/right)
glm::vec2 l1(0, 0), l2(0, 1), l3(1, 1), l4(1, 0);

// UV για τη σκεπή
glm::vec2 rA(0, 0), rB(1, 0), rC(0.5f, 1.0f);

// front z=0
triPN(verts, norms, v1, v2, v3); tri_uv(uvs, t1, t2, t3);
triPN(verts, norms, v1, v3, v4); tri_uv(uvs, t1, t3, t4);
```

Για τα πλευρικά (left/right) χρησιμοποιούνται uv που τρέχουν κατά μήκος του ύψους y και του βάθους z.

- Για τη σκεπή χρησιμοποιούμε τριγωνικό mapping με σημεία (0,0), (1,0), (0.5,1) ώστε να γεμίζει όλο το τρίγωνο:

```
// roof
triPN(verts, norms, v4, v3, v9); tri_uv(uvs, rA, rB, rC);
```

Επισήμανση! → Αναφέρεται προς το τέλος η λειτουργία της συνάρτησης triPN(…).

Φόρτωση υφής

- Χρησιμοποιείται η βιβλιοθήκη stb_image.h για τη φόρτωση της εικόνας textureA.jpg:

```
// Φόρτωση textureA.jpg
GLuint LoadTexture(const char* filename)
{
    int w, h, n;
    stbi_set_flip_vertically_on_load(1);
    unsigned char* data = stbi_load(filename, &w, &h, &n, 0);
    if (!data) {
        std::cerr << "Failed to load texture: " << filename << std::endl;
        return 0;
    }

    GLenum format = (n == 3) ? GL_RGB : GL_RGBA;

    GLuint tex;
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexImage2D(GL_TEXTURE_2D, 0, format, w, h, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
```

Το texture συνδέεται με το sampler2D του fragment shader:

```
glUniform1i(UseTexID, 1);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texA);
glUniform1i(TexID, 0);
glUniform4f(ColorID, 1.0f, 1.0f, 1.0f, 1.0f);
```

Κίνηση στον άξονα x (J/L)

- Η μετατόπιση του A αποθηκεύεται στο AoffsetX. Κάθε πάτημα του J/L μετακινεί τον χαρακτήρα κατά a/2:

```
// Κίνηση χαρακτήρα A (edge trigger) αν δεν έχει τελειώσει το παιχνίδι και δεν είνα
if (!gameOver && !paused) {
    bool L = glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS;
    bool J = glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS;
    if (L && !pL) AoffsetX += Astep;
    if (J && !pJ) AoffsetX -= Astep;
    pL = L; pJ = J;
}
```

Η τελική θέση περνά στο μοντέλο μέσω:

```
// Character A (αν δεν έχει gameOver)
if (!gameOver) {
    glm::mat4 ModelA = glm::translate(glm::mat4(1.0f), glm::vec3(AoffsetX, 0, 0));
    glm::mat4 MVP_A = P * V * ModelA;
```

(iii) Απαιτήσεις - Κύβοι-εχθροί και κίνηση προς τα κάτω

Οι εχθροί είναι 5 κύβοι με μέγεθος ENEMY_SIZE = 2.0f και half-dimensions EnemyHalf(1.0f)· η γεωμετρία κύβου παράγεται στη build_box_center.

- Οι θέσεις των κέντρων αποθηκεύονται σε πίνακα glm::vec3 enemyPos[ENEMY_COUNT]; και η κατάσταση (ζωντανός/νεκρός) σε bool enemyAlive[ENEMY_COUNT];.

- Το βασικό βήμα καθόδου στον y άξονα είναι:

const float ENEMY_STEP = ENEMY_SIZE * 0.5f;

Η κάθοδος γίνεται κάθε enemyStepPeriod δευτερόλεπτα μέσα στο main loop, στην φάση 4 του μοτίβου κίνησης:

```
// Εχθροί: μοτίβο κίνησης (γ)
// κάθε enemyStepPeriod δευτερόλεπτα κάνουν ΕΝΑ "βήμα" στο μοτίβο
if (!gameOver && (now - lastEnemyStep) >= enemyStepPeriod) { // αν δεν έχεις enemyS
    lastEnemyStep = now;
main(void)
    for (int i = 0; i < ENEMY_COUNT; ++i) {
        if (!enemyAlive[i]) continue;

        switch (enemyPhase) {
        case 0: // 5 βήματα δεξιά
            enemyPos[i].x += ENEMY_STEP_X;
            break;
        case 1: // 5 βήματα πίσω προς την αρχική θέση από δεξιά
            enemyPos[i].x -= ENEMY_STEP_X;
            break;
        case 2: // 5 βήματα αριστερά
            enemyPos[i].x -= ENEMY_STEP_X;
            break;
        case 3: // 5 βήματα πίσω προς την αρχική θέση από αριστερά
            enemyPos[i].x += ENEMY_STEP_X;
            break;
        case 4: // 1 βήμα προς τα κάτω στον y άξονα
            enemyPos[i].y -= ENEMY_STEP;
            break;
        }
```

Αν η κάτω πλευρά ενός οποιουδήποτε ζωντανού εχθρού φτάσει ή περάσει το ύψος του χαρακτήρα A (A_yTop), το παιχνίδι τελειώνει:

```
float enemyBottomY = enemyPos[i].y - EnemyHalf.y;
if (enemyBottomY <= A_yTop) {
    gameOver = true;
```

(iv) Απαιτήσεις- Βλήματα

Η γεωμετρία του βλήματος είναι παραλληλεπίπεδο 0.25 x 0.5 x 0.25, με half-dimensions:

const glm::vec3 BulletHalf(0.25f * 0.5f, 0.5f * 0.5f, 0.25f * 0.5f);

και παράγεται επίσης με build_box_center. Κάθε βλήμα αποθηκεύεται σε struct:

```cpp
// bullets
struct Bullet {
    glm::vec3 pos;
    bool active;
};
```

std::vector<Bullet> bullets;

Πυροδότηση με SPACE (ένα βλήμα ανά πάτημα – edge detect):

```cpp
// Πυροβολισμός με SPACE (ένα βλήμα ανά πάτημα)
bool Space = glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS;
if (!gameOver && !paused && Space && !pSpace) {
    Bullet b;
    b.pos = glm::vec3(AoffsetX, 0, 0) + A_apex_local; // v9 σε world space
    b.active = true;
    bullets.push_back(b);
}
pSpace = Space;
```

Η κίνηση του βλήματος είναι σταθερή στον +y άξονα:

const float BulletSpeed = 8.0f;

```cpp
// Ενημέρωση βλημάτων (πάνω στον +y)
if (!gameOver && !paused) {
    for (auto& b : bullets)
        if (b.active)
            b.pos.y += BulletSpeed * dt;
}
```

Έλεγχος σύγκρουσης βλήματος–εχθρού

- Χρησιμοποιείται απλό AABB collision σε 2D (x,y). Για κάθε ενεργό βλήμα υπολογίζονται τα min/max σε x,y, και συγκρίνονται με τα αντίστοιχα του εχθρού:

```cpp
// αγνοούμε το z -> 2D σύγκρουση (x,y)
if (overlap1D(bMinX, bMaxX, eMinX, eMaxX) &&
    overlap1D(bMinY, bMaxY, eMinY, eMaxY)) {
    enemyAlive[i] = false;    // ο εχθρός καταστρέφεται
    b.active = false;         // σταματάμε το βλήμα
    break;
}
```

Η overlap1D ελέγχει αν δύο διαστήματα [aMin,aMax], [bMin,bMax] τέμνονται.

(v) Απαιτήσεις- Κάμερα

Εδώ ο κώδικας είναι ίδιος με τον κώδικα της 1Β. Έχουμε εξηγήσει αναλυτικά της δημιουργία και την λειτουργία της κάμερας στην αναφορά της άσκησης 1Β. Μία σημαντική υπενθύμιση είναι πως χρησιμοποιούμε quaternions για ομαλή σύνθεση περιστροφών:

```
// --- Προσανατολισμός με quaternions ---
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;

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

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

gCameraPos = eye;
```

BONUS **

(α) Απαιτήσεις- Οπτικά & ηχητικά εφέ στο game over

Όταν ένας εχθρός φτάσει τον χαρακτήρα Α, ενεργοποιείται game over, σταματούν τα βλήματα. Σε Windows, παίζει αρχείο ήχου με την κλήση:

```
// === ΗΧΗΤΙΚΟ ΕΦΕ (μόνο σε Windows) ===

PlaySound(TEXT("new-notification-09-352705.mp3"), NULL, SND_FILENAME | SND_ASYNC);


// === ΟΠΤΙΚΟ ΕΦΕ: ενεργοποίηση κόκκινου flash ===
gameOverFlash = 1.0f;

break;
```

Υλοποιείται οπτικό εφέ «κόκκινο flash» μέσω μεταβλητής gameOverFlash που μειώνεται με το χρόνο, επηρεάζοντας το glClearColor. Πιθανόν να χρειαστεί στο Visual Studio η παρα-κάτω διαδικασία:
Project Properties → *Linker* → *Input* → *Additional Dependencies* και προσθήκη winmm.lib. Σε εμάς δεν χρειάστηκε αλλά το παρατηρήσαμε όταν το "τρέξαμε" σε άλλον υπολογιστή. Επιπλέον, πρέπει το αρχείο mp3 πρέπει να βρίσκεται στον ίδιο φάκελο με το εκτελέσιμο ή σε path ορατό από τη DLL του WinMM.

```
#ifdef _WIN32
#include <windows.h>
#include <mmsystem.h>
#pragma comment(lib, "winmm.lib")
#endif
```

(β) Απαιτήσεις-Πλήκτρα ειδικών λειτουργιών (R, P, F, S)

Μεταβλητές:

P – Pause/Unpause:
Edge-detect πάνω στο πλήκτρο P, απλά κάνει toggle της paused:

```
// --- (β) Pause / Restart / Ταχύτητα εχθρών ---

bool paused = false;                // αν είναι true, παγώνουν οι κινήσεις
bool prevP = false, prevR = false;  // για edge-detect στα P, R
bool prevF = false, prevS = false;  // για edge-detect στα F, S

double enemyStepPeriod = 5.0;       // αρχικά 5 δευτ. ανά βήμα (όπως στην εκφώνηση)
```

```
// P: toggle pause (edge-detected)
if (keyP && !prevP) {
    paused = !paused;
}
prevP = keyP;
```

Όταν paused == true, δεν ενημερώνονται ούτε εχθροί, ούτε βλήματα.

R – Restart: επαναφέρει τα πάντα στην αρχική κατάσταση

- gameOver = false; paused = false;
- καθάρισμα bullets.clear();
- επαναφορά AoffsetX και enemyPos[i] στην αρχική διάταξη,
- reset του lastEnemyStep και του gameOverFlash.

F – αύξηση ταχύτητας καθόδου:

```
// F: αύξηση ταχύτητας καθόδου (μικρότερο χρονικό διάστημα ανά βήμα)
if (keyF && !prevF) {
    enemyStepPeriod *= 0.7;      // 30% πιο γρήγορα
    if (enemyStepPeriod < 1.0)   // όριο ασφαλείας
        enemyStepPeriod = 1.0;
}
prevF = keyF;
```

S – μείωση ταχύτητας

```
// S: μείωση ταχύτητας καθόδου (μεγαλύτερο χρονικό διάστημα ανά βήμα)
if (keyS && !prevS) {
    enemyStepPeriod *= 1.3;      // ~30% πιο αργά
    if (enemyStepPeriod > 10.0)  // όριο ασφαλείας
        enemyStepPeriod = 10.0;
}
prevS = keyS;
```

(γ) Απαιτήσεις- Μοτίβο κίνησης εχθρών (δεξιά–αριστερά–κάτω)

Χρησιμοποιούνται δύο μεταβλητές:

```
// --- (γ) Μοτίβο κίνησης εχθρών: δεξιά, πίσω, αριστερά, πίσω, κάτω ---
int   enemyPhase = 0;        // 0: δεξιά, 1: πίσω από δεξιά, 2: αριστερά, 3: πίσω από αρ
int   phaseStepCount = 0;        // μετράει πόσα "βήματα" έχουμε κάνει στην τρέχουσα φάσ
const int   stepsPerSide = 5;     // 5 βήματα όπως ζητά η εκφώνηση
const float ENEMY_STEP_X = ENEMY_SIZE * 0.5f;  // οριζόντιο βήμα (ίδιο μέγεθος με ENEMY_S
```

Κάθε enemyStepPeriod δευτερόλεπτα γίνεται ένα «βήμα». Ανάλογα με την enemyPhase:

- 0: οι εχθροί κινούνται 5 βήματα δεξιά,
- 1: 5 βήματα πίσω προς την αρχική θέση (αριστερά),
- 2: 5 βήματα αριστερά,
- 3: 5 βήματα πίσω προς την αρχική,
- 4: ένα βήμα κάτω στον y άξονα,
  μετά η φάση επιστρέφει στο 0.
- Έτσι υλοποιείται το ζητούμενο μοτίβο: δεξιά → πίσω → αριστερά → πίσω → κάτω, επανάληψη.

(δ) Απαιτήσεις- Φωτισμός Phong

Αρχικά να αναφέρουμε πως το φαινόμενο Phong είναι μια μέθοδος για να υπολογίζουμε πώς φωτίζεται ένα αντικείμενο σε μια 3D σκηνή. Ο στόχος είναι να κάνουμε την επιφάνεια να δείχνει ρεαλιστικά φωτισμένη. Το Phong αποτελείται από τρία μέρη:

1. Ambient (περιβάλλον)

Είναι το "φως του δωματίου". Υπάρχει παντού, δεν έχει κατεύθυνση, και ορίζει το βασικό φωτεινό επίπεδο.

2. Diffuse (διάχυση)

Είναι το φως που πέφτει πάνω σε μια επιφάνεια και "σκορπίζεται". Εξαρτάται από τη γωνία ανάμεσα στη normal της επιφάνειας και τη φωτεινή πηγή. Αν η επιφάνεια κοιτάει προς το φως → φωτεινή. Αν είναι κάθετη → πιο σκοτεινή. Αν είναι ανάποδα → καθόλου φωτισμένη

3. Specular (ανακλάσεις / λάμψη)

Είναι η "γυαλάδα" που βλέπουμε σε κάποιο μέταλλο, αυτοκίνητο, πλαστικό κτλ. Εξαρτάται από τη normal της επιφάνειας, τη θέση της κάμερας, τη θέση της πηγής φωτός, το "shininess" υλικό (πόσο γυαλιστερό είναι κάτι).

ΤΕΛΙΚΟ ΦΩΣ PHONG = ambient + diffuse + specular

Στο πρόγραμμά μας αυτό γίνεται στον fragment shader ο οποίος θα αναλυθεί στη συνέχεια.

Τα normals είναι διανύσματα που δείχνουν κάθετα στην επιφάνεια ενός τριγώνου ή μιας κορυφής. Παράδειγμα, αν ένα τρίγωνο κοιτάει προς τα “εξωτερικά”, το normal δείχνει προς τα έξω. Χωρίς normals, ο φωτισμός δεν μπορεί να υπολογιστεί. Ο diffuse φωτισμός υπολογίζεται με:

diff = max(dot(N, L), 0.0), όπου N = το normal και L = κατεύθυνση φωτός → σημείο. Αν N και L σχηματίζουν μικρή γωνία → φωτεινό. Αν σχηματίζουν 90° → σκοτεινό. Αν είναι αντίθετα → μαύρο. Το specular επίσης χρειάζεται το normal για να υπολογίσει τις γυαλάδες. Στην υλοποίηση μας τα normals παράγονται ως εξής:

1. Παίρνουμε 2 πλευρές του τριγώνου: B - A, C - A
2. Παίρνουμε το cross product: cross(B - A, C – A).Αυτό δίνει ένα διάνυσμα κάθετο στην επιφάνεια.
3. Το κανονικοποιούμε (unit vector): normalize(...)
4. Το αποθηκεύουμε 3 φορές (1 για κάθε κορυφή).

Προστέθηκαν κανονικές (normals) τόσο στον χαρακτήρα A όσο και στα κουτιά/βλήματα μέσω της συνάρτησης triPN και των τροποποιημένων build_A, build_box_center.

- Νέα uniforms:

```
GLuint ModelID = glGetUniformLocation(programID, "M");
GLuint NormalMatID = glGetUniformLocation(programID, "NormalMatrix");
GLuint ViewPosID = glGetUniformLocation(programID, "viewPos");
GLuint LightPosID = glGetUniformLocation(programID, "lightPos");
```

Για κάθε αντικείμενο υπολογίζεται:

glm::mat4 M;

glm::mat3 N = glm::mat3(glm::transpose(glm::inverse(M))); (πχ:)

```
glm::mat4 MVP_A = P * V * ModelA;
glm::mat3 NormalA = glm::mat3(glm::transpose(glm::inverse(ModelA)));
```

και στέλνεται στο shader.

- Η θέση φωτεινής πηγής αρχικά είναι (8.0, 0.0, 0.0) όπως ζητά η εκφώνηση.
- Στον fragment shader υλοποιείται το μοντέλο Phong με ambient, diffuse και specular όρους.

(ε) Απαιτήσεις – Μετακίνηση φωτεινής πηγής

Η θέση της πηγής αποθηκεύεται στο global:

glm::vec3 gLightPos(8.0f, 0.0f, 0.0f);

Η κίνηση γίνεται συνεχώς με βάση το dt, αντίστοιχα με την κάμερα:

- LEFT/RIGHT: μεταβολή στον x άξονα,
- UP/DOWN: μεταβολή στον z άξονα,
- PAGE_UP/PAGE_DOWN: μεταβολή στον y άξονα.

```
// Χρησιμοποιούμε arrow keys + PageUp/PageDown ακριβώς όπως την κάμερα:

if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
    gLightPos.x -= lightMoveSpeed * dt;  // προς τα αριστερά
}
```

Η τρέχουσα θέση στέλνεται κάθε frame στο shader

```
glUniform3fv(LightPosID, 1, &gLightPos.x); // (8,0,0) όπως ορίσαμε global
```

(στ) Απαίτησεις – Μοντέλα OBJ

Το ερώτημα αυτό δεν υλοποιήθηκε στον τελικό κώδικα.

Η συνάρτηση triPN(…) κάνει τα εξής:

1. Προσθέτει ένα τρίγωνο στη λίστα κορυφών

Οι τρεις κορυφές A, B, C μπαίνουν στο verts.push_back(...).

2. Υπολογίζει το normal του τριγώνου ΚΑΙ το αποθηκεύει

Με αυτή τη γραμμή: glm::vec3 n = glm::normalize(glm::cross(B - A, C – A));

υπολογίζει το διάνυσμα normal του επίπεδου τριγώνου. Αυτό το normal προστίθεται τρεις φορές, μία για κάθε κορυφή του τριγώνου:

norms.push_back(n);

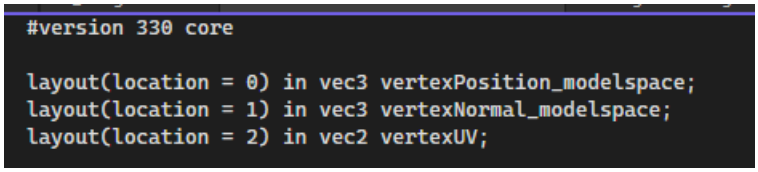
norms.push_back(n);

norms.push_back(n);

Χωρίς αυτή τη διαδικασία το μοντέλο δεν θα έχει σωστό φωτισμό. Χρειαζόμαστε αυτή την συνάρτηση καθώς το Phong lighting απαιτεί normals. Κάθε τρίγωνο χρειάζεται ένα normal για να ξέρει "προς ποια κατεύθυνση κοιτάει", ώστε να υπολογιστεί σωστά πόσο φως δέχεται (diffuse) και πόση λάμψη θα έχει (specular).

Τέλος, παραθέτουμε την εξήγηση για τα αρχεία P1CFragmentShader.fragmentshader και P1CVertexShader.vertexshader.

Ο VertexShader συνοπτικά κάνει τα εξής μετατρέπει κορυφές σε world space, μετατρέπει normals σωστά, περνά UV, υπολογίζει τη θέση στην οθόνη, στέλνει FragPos, Normal, UV στον fragment shader.



```
#version 330 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexNormal_modelspace;
layout(location = 2) in vec2 vertexUV;
```

Παραπάνω δηλώνουμε τα 3 attributes που δίνουμε από το αρχείο Source-1C:

location	**δεδομένο**	**σημασία**
0	θέση κορυφής	3D σημείο στον τοπικό χώρο του αντικειμένου
1	normal κορυφής	διάνυσμα κάθετο στην επιφάνεια
2	UV συντεταγμένες για το texture mapping	

Αυτά τα στείλαμε με glVertexAttribPointer(...) στο C++. Τώρα πρέπει να δηλώσουμε τις μεταβλητές που θα περάσουν στον Fragment Shader:

```
out vec3 FragPos;
out vec3 Normal;
out vec2 UV;
```

Τα παρακάτω uniforms τα δίνουμε από το Source-1C με glUniformMatrix...

- MVP χρησιμοποιείται για να τοποθετηθεί η κορυφή στην οθόνη
- M μετατρέπει από local space → world space
- NormalMatrix μετατρέπει σωστά τα normals (ώστε να μην "στραβώνουν" όταν έχεις scaling).

```
uniform mat4 MVP;
uniform mat4 M;
uniform mat3 NormalMatrix;
```

Μέσα στην main παίρνουμε τη θέση της κορυφής, την πολλαπλασιάζουμε με το Model και κατόπιν την στέλνουμε στον fragment shader ως FragPos. Ο φωτισμός Phong χρειάζεται world-space θέσεις. Έπειτα, κάνουμε την μετατροπή normal που είναι απαραίτητη διότι αν το μοντέλο έχει scaling, οι normals παραμορφώνονται.
Η NormalMatrix διορθώνει αυτή την παραμόρφωση. Τα UV παραμένουν ίδια. Τέλος έχουμε την τελική θέση στην οθόνη.

```
void main() {
    vec4 worldPos = M * vec4(vertexPosition_modelspace, 1.0);
    FragPos = worldPos.xyz;
    Normal  = normalize(NormalMatrix * vertexNormal_modelspace);
    UV      = vertexUV;

    gl_Position = MVP * vec4(vertexPosition_modelspace, 1.0);
}
```

Ο Fragment Shader συνοπτικά κάνει τα εξής: Παίρνει χρώμα από υφή ή από uniform, Υπολογίζει τα vectors για το Phong (N, L, V, R), Υπολογίζει Ambient - Diffuse - Specular φωτισμό και συνδυάζει τα 3 και βγάζει το τελικό χρώμα.

Λαμβάνει από τον vertex shader:

```
#version 330 core

in vec3 FragPos;
in vec3 Normal;
in vec2 UV;
```

Τα Uniforms σημαίνουν:

- useTexture: αν 1 → χρησιμοποίησε την υφή, αν 0 → χρησιμοποίησε το σταθερό χρώμα uniColor
- lightPos → πού βρίσκεται η φωτεινή πηγή

- viewPos → πού βρίσκεται η κάμερα

```
uniform sampler2D texSampler;
uniform vec4  uniColor;
uniform int   useTexture;

uniform vec3  lightPos;   // (8,0,0) από C++
uniform vec3  viewPos;    // θέση κάμερας από C++
```

Το παρακάτω screenshot δείχνει πως αν έχουμε texture τότε παίρνουμε χρώμα από υφή, αλλιώς παίρνουμε από uniform color.

```
// Βασικό χρώμα: από υφή ή από σταθερό χρώμα
vec3 baseColor = (useTexture == 1)
    ? texture(texSampler, UV).rgb
    : uniColor.rgb;
```

Έπειτα γίνεται η κανονικοποίηση των διανυσμάτων N: normal, L: διάνυσμα προς το φως, V: διάνυσμα προς την κάμερα, R: ανακλώμενη ακτίνα για specular που χρειάζονται στο Phong.

```
vec3 N = normalize(Normal);
vec3 L = normalize(lightPos - FragPos);
vec3 V = normalize(viewPos - FragPos);
vec3 R = reflect(-L, N);
```

Υπολογίζεται ο όρος ambient με: vec3 ambient = 0.15 * baseColor; Ο όρος diffuse υπολογίζεται στις γραμμές float diff = max(dot(N, L), 0.0) και vec3 diffuse = diff * baseColor; Τέλος, ο όρος specular υπολογίζεται στις γραμμές 30-31 (στο if). Το pow(..., 32) είναι το shininess. Με το dot(R,V) δείχνει πόσο βλέπουμε μία γυαλιστερή αντανάκλαση και το λευκό specular είναι (vec3(1.0)).

Το τελικό χρώμα προκύπτει από την πρόσθεση των επιμέρους όρων στις γραμμες 37-38.

ΔΥΣΚΟΛΙΕΣ

1) Αρχικά υπήρχαν προβλήματα με την φόρτωση της υφής (λάθος path / working directory). Λύθηκε τοποθετώντας το textureA.jpg στον φάκελο του εκτελέσιμου και με σωστή χρήση της stb_image.

2) Για τον φωτισμό Phong χρειάστηκε τροποποίηση της υπάρχουσας γεωμετρίας (παλιές συναρτήσεις tri(..) που έγινε triPN(..)) ώστε να υπολογίζονται και normals για κάθε τρίγωνο.

3) Η λογική pause/restart και το μοτίβο κίνησης των εχθρών ήταν σχετικά σύνθετα λόγω των πολλών καταστάσεων (phase, step counters, enemyStepPeriod).

2. Πληροφορίες σχετικά με την υλοποίηση

Η εργασία υλοποιήθηκε σε περιβάλλον Windows, χρησιμοποιώντας Visual Studio και βιβλιοθήκες OpenGL 3.3. Παρακάτω παρουσιάζονται οι βασικές τεχνικές λεπτομέρειες που

απαιτούνται για τη μεταγλώττιση, εκτέλεση και κατανόηση του κώδικα. Χρησιμοποιούμε τις παρακάτω βιβλιοθήκες:GLFW, GLEW, GLM, stb_image.h, WinMM (για ήχο) η οποία απαιτεί το #pragma comment(lib, "winmm.lib"). Χρησιμοποιούνται custom vertex και fragment shaders, οι οποίοι υλοποιούν προβολή & μετασχηματισμούς (MVP), υφή (texture sampling), Phong φωτισμό (ambient, diffuse, specular) και υποστήριξη για normals. Για την σωστή μεταγλώττιση απαιτούνται τα εξής: εισαγωγή όλων των .cpp, .h και shader files στο Visual Studio, τα απαιτούμενα αρχεία στο ίδιο directory με το .exe (textureA.jpg, new-notification-09-352705.mp3, P1CVertexShader.vertexshader, P1CFragmentShader.fragmentshader), η ενεργοποίηση του winmm.lib για τον ήχο. Η αναπαραγωγή ήχου λειτουργεί μόνο σε Windows. Τα normals παράγονται με βάση το cross product στο helper triPN.

3. Σύντομη αξιολόγηση της λειτουργίας της ομάδας σας και της συνεργασίας

Η εργασία υλοποιήθηκε με συστηματική συνεργασία, ανταλλαγή κώδικα και κοινή προσπάθεια debbugging. Η συνεργασία ήταν αποδοτική και ισορροπημένη. Κάθε μέλος ανέλαβε διαφορετικά κομμάτια της λογικής και της υλοποίησης ενώ η τελική ένωση του κώδικα έγινε με κοινή επίβλεψη. Προέκυψαν τεχνικές δυσκολίες, κυρίως με τη φόρτωση υφών και την ορθή κατασκευή των normals, οι οποίες επιλύθηκαν με κοινή προσπάθεια.

Φοιτητής 5233

Ρύθμιση παραθύρου, GLEW/GLFW αρχικοποίηση. Υλοποίηση εχθρών, γεωμετρίας κύβων και κίνησης κατά τον άξονα y. Πλήρης ανάπτυξη του μοτίβου κίνησης (δεξιά–πίσω–αριστερά–πίσω–κάτω). Υλοποίηση pause, restart, επιτάχυνσης/επιβράδυνσης εχθρών (P, R, F, S). Αναπαραγωγή ήχου και οπτικό κόκκινο flash στο game over. Collision detection με βλήματα. Debugging σε shaders και lighting.

Φοιτήτρια 5386

Σχεδίαση και υλοποίηση γεωμετρίας χαρακτήρα A με σωστές αναλογίες. Υπολογισμός UV χαρτογράφησης (texture mapping) σε όλες τις πλευρές. Ενσωμάτωση υφής μέσω της βιβλιοθήκης stb_image.h. Υλοποίηση orbit κάμερας με quaternions και smooth dt movement. Υλοποίηση πλήρους φωτισμού Phong (ambient + diffuse + specular). Μετακίνηση φωτεινής πηγής με 6 πλήκτρα (← → ↑ ↓ PgUp PgDown).

Η συνεργασία έγινε με ανταλλαγή κώδικα σε μορφή zip και κοινό έλεγχο μετά από κάθε αλλαγή. Χωρισμός της εργασίας σε modules (γεωμετρία, κάμερα, εχθροί, φωτισμός). Συνεργατικό debugging σε shaders, ιδιαίτερα στο κομμάτι phong lighting και UV mapping και συχνές συναντήσεις για συγχώνευση αλλαγών και δοκιμές στο κοινό project.

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

- Για την stb_image.h – https://github.com/nothings/stb/blob/master/stb_image.h

  https://stackoverflow.com/questions/1128988/c-cross-platform-image-loader-for-opengl

- Για την PlaySound function και winmm.lib. - https://www.youtube.com/watch?v=CrPHVvgENq0

  https://learn.microsoft.com/en-us/previous-versions/dd743680(v=vs.85)

- Διαφάνειες εργαστηρίου και διαλέξεων

- ChatGPT – βοήθεια σε debbugging και οργάνωση κώδικα. Χρησιμοποιήθηκε κυρίως για διόρθωση shader, και τη δομή του game loop.