

# Προγραμματιστικές Ασκήσεις OpenGL 2024-25

## 3ο μέρος

Χαλαϊδόπουλος Μιλτιάδης AM 5384  
Ηλιάκης Ηλίας AM 5228

18/11/2024 - 2/12/2024

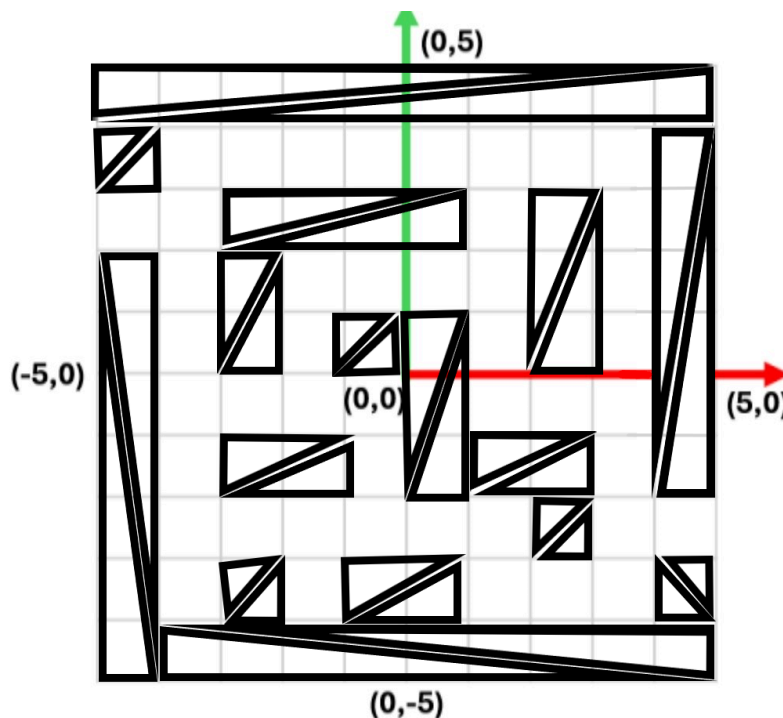
**1ο ερώτημα:** Δουλεύουμε παρόμοια με τις άλλες 2 ασκήσεις, αλλάζουμε τις συντεταγμένες του παραθύρου, βάζουμε τον τίτλο του παραθύρου να δέχεται ελληνικούς χαρακτήρες (το `u8` πριν από τον τίτλο), στην συνέχεια αλλάζουμε το background color σε μαύρο (`rgb = 000`), τέλος στην συνθήκη του κύριου Loop της main το (το `do-while`) αλλάζουμε την συνθήκη τερματισμού ώστε να τερματίζει πλέον με το πάτημα του `space`.

```
//Δημιουργία παραθύρου  
window = glfwCreateWindow(950, 950, u8"Άσκηση 1B-2024 ", NULL, NULL);
```

```
// background color  
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

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

**2ο ερώτημα:** Για την σχεδίαση του λαβυρινθου χρησιμοποιησαμε το πρότυπο της εκφωνησης της ασκήσεις και σχηματισαμε τα ορθογωνια και τα τετραγωνα του λαβυρινθου συμφωνα με το παρακάτω διάγραμμα, πρωτα για το επιπεδο  $z=0$  από την προηγουμενη ασκηση, και επειτα στο επιπεδο  $z=1$  και μετα αντίστοιχα για καθε τοιχο του λαβυρινθου αντιστοιχουσαν δυο τριγωνα, ετσι ωστε καθε κυβος/ορθογωνιο να εχει στο συνολο 12 τριγωνα :



Για να γίνει αυτό έπρεπε να βάλουμε τις κατάλληλες συντεταγμένες για κάθε τρίγωνο που θα ζωγραφιστει στον shape buffer του λαβυρίνθου( ο `cube[]` ). Μετά τροποποιούμε την συνάρτηση που ζωγραφίζει τα τρίγωνα σε : `glDrawArrays(GL_TRIANGLES, 0, 468)`; έτσι ώστε να ζωγραφίσει όλα τα απαραίτητα τρίγωνα. Η αλλαγή του χρώματος του λαβυρίνθου θα εξηγηθεί παρακάτω.

```
//buffer λαβύρινθου
static const GLfloat cube[] =
{
    -5.0f, -5.0f, 0.0f, //επίπεδο
    -4.0f, -5.0f, 0.0f,
    -5.0f, 2.0f, 0.0f, //3
    -4.0f, -5.0f, 0.0f,
    -4.0f, 2.0f, 0.0f,
    -5.0f, 2.0f, 0.0f, //6

    -4.0f, -5.0f, 0.0f,
    5.0f, -5.0f, 0.0f,
    5.0f, -4.0f, 0.0f,
    -4.0f, -5.0f, 0.0f,
    5.0f, -4.0f, 0.0f,
    -4.0f, -4.0f, 0.0f, //12

    4.0f, -4.0f, 0.0f,
    5.0f, -4.0f, 0.0f,
    5.0f, -3.0f, 0.0f,
    4.0f, -4.0f, 0.0f,
    5.0f, -3.0f, 0.0f,
    4.0f, -3.0f, 0.0f, //18

    -5.0f, 4.0f, 0.0f,
    -4.0f, 4.0f, 0.0f,
    -5.0f, 2.0f, 0.0f,
    -4.0f, 2.0f, 0.0f,
    -5.0f, 2.0f, 0.0f,
    -4.0f, 2.0f, 0.0f, //24
}
```

Με παρόμοιο τρόπο με τον λαβύρινθο δουλεύουμε και για την δημιουργία του κύβου ο οποίος θα τον διασχίζει. Ο Κύβος αυτός θα έχει κέντρο αρχικά το (-4.5,2.5,0.5) και πλευρές μήκους 0.5. Τα αρχικά σημεία του κύβου είναι τα (-4.75, 2.25,0.25 -4.25, 2.25,0.25 -4.25,2.75,0.25 -4.75,2.75,0.25 καθώς και άλλα 4 με ίδια x,y με τα προηγούμενα αλλά με z=0.75)(παρακάτω φαίνεται και το shape buffer του) για να αποτυπωθεί ο κύβος δημιουργούμε ένα νέο VBO το VBO2 και μία νέα συνάρτηση που ζωγραφίζει τα τρίγωνα (επειδή το νέο αντικείμενο θα αλλάζει θέση γίνεται bind δυναμικά μέσα στο κύριο loop της main). Τέλος μένει να γίνει αλλαγή των χρωμάτων τόσο του λαβύρινθου όσο και του κύβου που τον διασχίζει. Αυτό γίνεται αρκετά διαφορετικά από την πρώτη άσκηση καθώς τώρα πρέπει να δημιουργήσουμε έναν color buffer για καθένα από τα δύο μας αντικείμενα και να δώσουμε τα κατάλληλα χρώματα (μπλε στον λαβύρινθο rgb=001 και κίτρινο στον κύβο rgb=110).

Παρακάτω παρατίθεται μόνο ο color buffer για τον κύβο ωστόσο αυτός για τον λαβύρινθο είναι παρόμοιος και υπάρχει στον κώδικα της άσκησης.

```
//buffer χαρακτήρα
static GLfloat shape_2_buffer[]
    -4.75f, 2.25f, 0.25f, // κ
    -4.25f, 2.25f, 0.25f, //3
    -4.25f, 2.75f, 0.25f, //6
    -4.75f, 2.25f, 0.25f, //9
    -4.25f, 2.75f, 0.25f,
    -4.75f, 2.75f, 0.25f,

    -4.75f, 2.25f, 0.75f, // π
    -4.25f, 2.25f, 0.75f,
    -4.25f, 2.75f, 0.75f,
    -4.75f, 2.25f, 0.75f,
    -4.25f, 2.75f, 0.75f,
    -4.75f, 2.75f, 0.75f, //33

    -4.75f, 2.25f, 0.25f, //μπρ
    -4.25f, 2.25f, 0.25f,
    -4.25f, 2.25f, 0.75f,
    -4.75f, 2.25f, 0.25f,
    -4.25f, 2.25f, 0.75f,
    -4.75f, 2.25f, 0.75f, //45

    -4.75f, 2.75f, 0.25f, // α
    -4.75f, 2.25f, 0.25f,
```

```
static const GLfloat colortwo[]
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
    1.000f, 1.000f, 0.000f, a,
```

*shape buffer (αριστερά) και color buffer του κύβου*

Στο σημείο αυτό πρέπει να φτιάξουμε την κίνηση του τετραγώνου μέσα στον λαβύρινθο. Η βασική ιδέα είναι ότι κάθε φορά που ο χρήστης θα πατάει ένα από τα κουμπιά IJKL το τετράγωνο θα μετακινείται μέσα στο λαβύρινθο μεταβάλλοντας κάθε φορά τις συντεταγμένες του x ή y κατά 1 ή -1 (ανάλογα με ποιο πλήκτρο πατήθηκε). Πρώτα από όλα θα θέλαμε όταν πατιέται το κουμπί L ο χαρακτήρας να μετακινείται κατά μία θέση δεξιά, με άλλα λόγια το X του κέντρου του να αυξάνεται κατά 1. Αυτό μπορεί να επιτευχθεί αν η συντεταγμένη X κάθε μίας από τις ακμές του αυξηθεί κατά 1 (για αυτό το λόγο ενημερώνουμε όλα τα X του buffer αυξάνοντας τα) Όμοια επαναλαμβάνουμε και για όταν πατηθεί το κουμπί KJI αλλάζοντας κατάλληλα τις συντεταγμένες X ή Y. Αυτό όμως δεν είναι αρκετό, το κομμάτι αυτό του κώδικα που κάνει τον έλεγχο για το αν πατήθηκε κάποιο κουμπί βρίσκεται μέσα στο κύριο loop του προγράμματος, αυτό σημαίνει ότι το πρόγραμμα θα ελέγχει διαρκώς αν ένα πλήκτρο είναι πατημένο με αποτέλεσμα να μετακινεί τον χαρακτήρα παραπάνω από μια φορές αν ο χρήστης δεν έχει προλάβει να αφήσει το πλήκτρο έγκυρα. Πρέπει να διασφαλίσουμε ότι ένα πάτημα κάποιου πλήκτρου συνεπάγεται σε μετακίνηση του

χαρακτήρα κατά μία μόνο θέση. Για να συμβεί αυτό αρκεί να “κλειδώσουμε” το κομμάτι του κώδικα που είναι υπεύθυνο για την μετακίνηση του παίκτη όση ώρα κάποιο πλήκτρο είναι πατημένο. Αυτό γίνεται εύκολα με την χρήση μιας bool μεταβλητής, της stall. Ουσιαστικά αυτή η μεταβλητή έχει την λογική τιμή true όση ώρα υπάρχει κάποιο πλήκτρο μετακίνησης πατημένο και λογική τιμή false όλες τις άλλες περιπτώσεις. Ο χαρακτήρας μπορεί να μετακινηθεί μόνο όταν η μεταβλητή έχει την τιμή false. Είναι σημαντικό ο έλεγχος για αν ένα πλήκτρο κίνησης είναι πατημένο να γίνεται σε ύστερο σημείο στο loop από την κίνηση του χαρακτήρα έτσι ώστε να προλαβαίνει να μετακινηθεί κατά μία θέση προτού η μεταβλητή stall πάρει την τιμή true.

```
glBindBuffer(GL_ARRAY_BUFFER, VBO2);
glBufferData(GL_ARRAY_BUFFER, sizeof(shape_2_buffer), shape_2_buffer, GL_DYNAMIC_DRAW);
if (!stall) {
    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS) {
        shape_2_buffer[0] = shape_2_buffer[0] + 1;
        shape_2_buffer[2] = shape_2_buffer[2] + 1;
        shape_2_buffer[4] = shape_2_buffer[4] + 1;
        shape_2_buffer[6] = shape_2_buffer[6] + 1;
        shape_2_buffer[8] = shape_2_buffer[8] + 1;
        shape_2_buffer[10] = shape_2_buffer[10] + 1;
    }
}
```

```
if (!stall) {
    if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS) {
```

```
    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS || glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS || glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS) {
        stall = true;
    }
    else {
        stall = false;
    }
}
```

```
int cubeX = 0; // θέση X του κύβου αυτή την στιγμή
int cubeY = 2; // θέση Y του κύβου αυτή την στιγμή
static float rotationAngleX = 0.0f;
float distance = zPos;
GLboolean mapArray[10][10] = { //έγκυρες θέσεις μετακίνησης στον λαβύρινθο για να μην περνάει ο
    {GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE},
    {GL_TRUE, GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE, GL_TRUE, GL_TRUE},
    {GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE},
    {GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE}
};
```

Τέλος πρέπει να κάνουμε τον χαρακτήρα να μεταφέρεται από την αρχή στο τέλος του λαβυρίνθου όταν πατηθούν τα κατάλληλα κουμπιά, για αυτό μπορούμε να ελέγξουμε την θέση του στον λαβύρινθο με τις μεταβλητές που δημιουργήσαμε παραπάνω (cubeX, cubeY). Για την μεταφορά του τώρα αρκεί να αλλάξουμε τις συντεταγμένες X και Y των κορυφών του αναλόγως. Γνωρίζουμε ότι τα x και y κάθε σημείου του κύβου βρίσκονται στις θέσεις του πίνακα με διαφορά 3 οπότε εκμεταλλευόμενοι αυτό φτιάχνουμε μια for με βήμα 3. Η επανάληψη αυτή είναι υπεύθυνη για την ενημέρωση των σημείων του κύβου. Η συντεταγμένη z δεν χρειάζεται αλλαγή καθώς μένει σταθερή για οποιαδήποτε κίνηση του κίτρινου κύβου. Για να αποφύγουμε ότι ο χαρακτήρας θα περνάει μέσα από τοίχους

δημιουργήσαμε έναν πίνακα με όλες τις έγκυρες θέσεις που μπορεί να βρεθεί, τον `mapArray`, και δύο μεταβλητές οι οποίες αποθηκεύουν την θέση του χαρακτήρα την δεδομένη στιγμή. Δηλαδή το `cubeX` και το `cubeY` του. Κάθε φορά που γίνεται μια νόμιμη μετακίνηση η θέση του παίκτη ενημερώνεται ενώ αν είναι μέσα σε τοίχο η μετακίνηση ακυρώνεται. Σημείωση: ο πίνακας παίρνει τιμές από 0 έως 10 και για το `x` και για το `y`. Το `x` αυξάνεται από αριστερά προς τα δεξιά ενώ το `y` από πάνω προς τα κάτω, έτσι η αρχική θέση του χαρακτήρα στον πίνακα είναι η  $(x,y)=0,2$ .

```
if (!stall) {
    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS) {
        if (cubeX == 9) { //εάν έφτασε στον τερματισμό και πατήθηκε το πλήκρο L
            int i;
            for(i = 0; i < 108; i+=3) {
                shape_2_buffer[i] = shape_2_buffer[i] - 9; //ενημέρωση των ακμών του κύβου
                shape_2_buffer[i + 1] = shape_2_buffer[i + 1] + 5;
            }
            cubeX = 0; //ενημέρωση ότι ο κύβος βρήσκειται πλέον στην αρχή
            cubeY = 2;
        }
        else { ... }
```

**3ο ερώτημα:** Σε αυτό το σημείο πρέπει να δημιουργήσουμε ένα 3ο , κυβικό αντικείμενο το οποίο θα είναι ο θησαυρός , θα έχει κέντρο το κέντρο του κάθε τετραγώνου στο οποίο βρίσκεται αλλά το μήκος κάθε πλευράς του θα είναι 0.8. Για αυτό το αντικείμενο όμοια με τον λαβύρινθο και τον χαρακτήρα δημιουργήσαμε ένα Vertex buffer (`shape_3_buffer[]`) και ένα color buffer (`colorthree[]`) ο οποίος αρχικοποιήθηκε στο χρώμα με `rgb=100` δηλαδή κόκκινο (δεν έχει και πολύ σημασία κυρίως το κάναμε για να έλεγχο του κώδικα αργότερα θα αναφερθεί και πως βάλαμε texture). Τέλος όπως και με τον χαρακτήρα που διασχίζει τον λαβύρινθο δημιουργήσαμε ένα 3ο Vertex buffer object (VBO3) (επειδή το νέο αντικείμενο θα αλλάζει θέση γίνεται `bind` δυναμικά μέσα στο κύριο loop της `main`). Τώρα πρέπει να κάνουμε τον θησαυρό να μετακινείται σε μια τυχαία έγκυρη θέση του λαβυρίνθου μετά από πέρασμα συγκεκριμένου χρόνου. Για αρχή για να βρούμε μια τυχαία θέση κάναμε χρήση της συνάρτησης `rand()` για αυτό τον λόγο χρειάστηκε στην `main` να αρχικοποιήσουμε το `seed` `srand(time(NULL))`; Για να βρούμε την τυχαία θέση πηγαίνουμε στο βασικό loop της `main` και προσθέτουμε κώδικα ο οποίος για κάθε επανάληψη θα βρίσκει μια τυχαία θέση μέσα στο λαβύρινθο, αν είναι έγκυρη θα μετακινεί εκεί τον θησαυρό αν όχι θα τον αφήνει στην θέση του , το ίδιο θα επαναλαμβάνεται συνεχόμενα. Ευτυχώς για εμάς έχουμε δημιουργήσει είδη μεταβλητές οι οποίες αποθηκεύουν ποιες θέσεις είναι νόμιμες και ποιές όχι, πιο συγκεκριμένα η `mapArray[i][j]== True` έχει όλες τις θέσεις του λαβυρίνθου στις οποίες δεν υπάρχει τοίχος και ο συνδυασμός `cubeY` και `cubeX` την τωρινή θέση του χαρακτήρα στον λαβύρινθο. Δημιουργούμε 2 νέες μεταβλητές τις `tresureX` και `tresureY` οι οποίες περιέχουν την τωρινή θέση του θησαυρού στο λαβύρινθο από 0 έως 10 σε ίδιο τρόπο αρίθμησης με τις αντίστοιχες `cubeX` και `cubeY`, τις αρχικοποιούμε ώστε να δείχουν το κέντρο της αρχικής θέσης στο λαβύρινθο  $(x,y = 1,2)$ , επιπλέον δημιουργήσαμε άλλες δύο μεταβλητές τις `newtresureX` και `newtresureY` οι οποίες χρησιμοποιούνται για τον έλεγχο της νέας θέσης του λαβυρίνθου. Έχοντας τα παραπάνω μπορούμε εύκολα να βρούμε την συνθήκη ελέγχου για έγκυρη θέση `if(mapArray[newtresureY][newtresureX] && newtresureY != cubeY && newtresureX != cubeX)` τώρα μετακινούμε τον χαρακτήρα σύμφωνα με τον παρακάτω κώδικα μέσα στην `if` :



```

for (j = 0; j < 108; j += 3) { //για κάθε 3άδα X,Y,Z
    shape_3_buffer[j] = shape_3_buffer[j] - trespureX + newtrespureX; //ενημέρωση των ακμών του θησαυρού
    shape_3_buffer[j + 1] = shape_3_buffer[j + 1] + trespureY - newtrespureY; //ενημέρωση ακμών
}
trespureX = newtrespureX; //αλλαγή θέσης X
trespureY = newtrespureY; //αλλαγή θέσης Y
newtrespureX = rand() % 10;
newtrespureY = rand() % 10;
} else {
    newtrespureX = rand() % 10;
    newtrespureY = rand() % 10;
}
}

```

Σημείωση: Οι `newtrespureX = rand() % 10;` γίνονται για να αλλάξει θέση τιμή η νέα θέση του θησαυρού και να μην “κολλήσει” σε μία συγκεκριμένη.

Αν τρέξουμε το πρόγραμμα τώρα θα παρατηρήσουμε ότι ο θησαυρός μετακινείται υπερβολικά γρήγορα με αποτέλεσμα να είναι αδύνατον να τον “ακουμπήσει” ο χαρακτήρας. Πρέπει να διασφαλίσουμε ότι ο θησαυρός μένει σε μία θέση για ένα χρονικό συγκεκριμένο χρονικό διάστημα προτού εξαφανιστεί. Αυτό γίνεται με την χρήση ενός flag του `lock` το οποίο παίρνει την τιμή 0 όταν είναι είναι ξεκλειδωμένο και ο κώδικας μπορεί να προχωρήσει κανονικά και 1 όταν είναι κλειδωμένο και το κομμάτι κώδικα εντός της `if` δεν εκτελείται (να σημειωθεί εδώ ότι η μέθοδος η οποία θα χρησιμοποιήσουμε για να μετριάσουμε την ταχύτητα μετακίνησης του θησαυρού δεν πρέπει να εμποδίζει την λειτουργία του υπόλοιπου `loop`). Η συνάρτηση κλειδώματος και ξεκλειδώματος φαίνεται παρακάτω:

```

int lock = 0; // Flag για κλείδωμα
clock_t lock_start_time; // Χρόνος έναρξης του κλειδώματος
int lock_duration = 2000; // Διάρκεια κλειδώματος σε milliseconds
void check_unlock() {
    if (lock && ((clock() - lock_start_time) * 1000 / CLOCKS_PER_SEC >= lock_duration)) {
        lock = 0; // Ξεκλείδωμα
    }
}

```

Έχοντας αυτό μπορούμε να κάνουμε τις τροποποιήσεις μέσα στο `loop` της `main` ώστε να περιμένει προτού μετακινηθεί ο χαρακτήρας. Παρατηρούμε ότι πέρα από την αναμενόμενη συνθήκη του `lock` στην `if` υπάρχει και μία Δεύτερη `lock2` αυτή είναι χρήσιμη για το επόμενο ερώτημα της άσκησης και θα εξηγηθεί εκεί η χρήση της.

Σημείωση: ο χρόνος ο οποίος θα μένει ο κύβος σε μία θέση είναι ίσος με 2 δευτερόλεπτα και ορίζεται από την μεταβλητή `lock_duration`.


```

//μετακίνηση θησαυρού
if (!lock && mapArray[newtrespureY][newtrespureX] && newtrespureY != cubeY && newtrespureX != cubeX && !lock2) {
    lock = 1; //κλείδωμα
    lock_start_time = clock(); // Καταγραφή χρόνου έναρξης του κλειδώματος όταν τελειώσει lock=0
    int j;
    for (j = 0; j < 108; j += 3) { //για κάθε 3άδα X,Y,Z
        shape_3_buffer[j] = shape_3_buffer[j] - trespureX + newtrespureX; //ενημέρωση των ακμών του θησαυρού
        shape_3_buffer[j + 1] = shape_3_buffer[j + 1] + trespureY - newtrespureY; //ενημέρωση ακμών
    }
    trespureX = newtrespureX; //αλλαγή θέσης X
    trespureY = newtrespureY; //αλλαγή θέσης Y
    newtrespureX = rand() % 10;
    newtrespureY = rand() % 10;
} else {
    newtrespureX = rand() % 10;
    newtrespureY = rand() % 10;
}
}

```

Το μόνο που μένει τώρα είναι να προσθέσουμε το `texture` στον θησαυρό. Πρώτα από όλα πρέπει να φορτώσουμε στο πρόγραμμα την εικόνα (η οποία είναι σε μορφή `.jpg`) για να γίνει αυτό χρησιμοποιούμε την έτοιμη βιβλιοθήκη που δόθηκε στο εργαστήριο `stb_image` με την κάλεσμα της συνάρτησης `data = stbi_load("coins.jpg", &w, &h, &comp, 0);`. Σημαντικό είναι

επίσης να υπολογίσουμε τις UV συντεταγμένες ώστε να γνωρίζει το πρόγραμμα που να δώσει το texture. Αυτές υπολογίζονται με αντιστοίχιση της εικονας coins.jpg απο (0,0)εως (1,1) με την πλευρα που θελουμε να δωσουμε texture την φορα.στον θησαυρο, η κάθε αντιστοιχία υπολογίζεται όπως και στο παραδειγμα texture mapping (πρόσθετες διαφάνειες του εργαστηρίου). Οποτε για το δεξι τριγωνο θα δωσουμε συντεταγμενες (0,0),(1,0),(1,1) και για το αριστερο (0,0),(1,1),(0,1) με αυτή τη σειρα για την μια πλευρα,οποτε αφου εχουμε 6 πλευρες συνολικα θα τις γραψουμε συνολικα 6 φορες ώστε στο συνολο να είναι 36.Τέλος κάνουμε τις απαραίτητες αλλαγές στους shaders. Πιο συγκεκριμένα ο Vertex Shader θα πάρει τις UV συντεταγμένες και θα τις περάσει στον fragment shader.Αυτό φαίνεται και παρακάτω.

 P1CVertexShader - Σημειωματάριο

Αρχείο Επεξεργασία Μορφή Προβολή Βοήθεια

```
#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;  
}
```





#### P1CFragmentShader - Σημειωματάριο

Αρχείο Επεξεργασία Μορφή Προβολή Βοήθεια

```
#version 330 core
in vec4 fragmentColor;
in vec2 UV;

out vec4 color;

uniform sampler2D myTextureSampler;
uniform bool uUseTexture = true;
void main()
{
    if(uUseTexture){
        color = fragmentColor;
    }
    else{
        color = texture(myTextureSampler,UV);
    }
}
```

**4ο ερώτημα:** Ο Θησαυρός πλέον μπορεί να κινείται ελεύθερα μέσα στον λαβύρινθο όταν όμως ο χαρακτήρας τον ακουμπήσει δεν γίνεται απολύτως τίποτα. Καταρχάς πρέπει να κάνουμε το μέγεθος του θησαυρού να μειώνεται στο μισό όταν τον ακουμπάει ο χαρακτήρας δηλαδή αν το `treasureX == cubeX && treasureY == cubeY` για αυτόν τον λόγο δημιουργούμε μια συνάρτηση η οποία θα μειώνει το μέγεθος του κύβου στο μισό χωρίς όμως να αλλάζει το κέντρο του. Η συνάρτηση υπολογίζει το κέντρο του κύβου υπολογίζοντας τον μέσο όρο κάθε συντεταγμένης και στην συνέχεια βρίσκει τα νέα vertices.

```
void halfCube(float* vertices) {
    float x_center = 0, y_center = 0, z_center = 0;

    // Υπολογισμός κέντρου
    for (int i = 0; i < 36; i++) {
        x_center += vertices[i * 3];
        y_center += vertices[i * 3 + 1];
        z_center += vertices[i * 3 + 2];
    }
    x_center /= 36;
    y_center /= 36;
    z_center /= 36;

    // Υπολογισμός νέων κορυφών
    for (int i = 0; i < 36; i++) {
        vertices[i * 3] = x_center + (vertices[i * 3] - x_center) * 0.5;
        vertices[i * 3 + 1] = y_center + (vertices[i * 3 + 1] - y_center) * 0.5;
        vertices[i * 3 + 2] = z_center + (vertices[i * 3 + 2] - z_center) * 0.5;
    }
}
```

Επιπλέον θέλουμε όταν ο χαρακτήρας πιάνει τον θησαυρό αυτός να σταματάει να μετακινείται, για αυτό τον λόγο δημιουργούμε ένα δεύτερο flag το `lock2` το οποίο είδαμε και

στο 2ο ερώτημα και όταν είναι διάφορο του 0 σταματάει την μετακίνηση του θησαυρού. Τέλος μετά από ένα χρονικό διάστημα πρέπει ο θησαυρός να εξαφανίζεται. Κάτι τέτοιο είναι εύκολο να γίνει αν χρησιμοποιήσουμε την lock2 που δημιουργήσαμε νωρίτερα με μια συνθήκη ελέγχου στην συνάρτηση που ζωγραφίζει τα τρίγωνα του θησαυρού. Αυτό φαίνεται και παρακάτω:

```
//ο χαρακτήρας "έπιασε" τον θησαυρό
if (lock2 == 0 && treasureX == cubeX && treasureY == cubeY) {
    lock2 = 1; //κλειδώνει
    lock_start_time2 = clock(); //παραμένει lock=1 για 5 δευτερόλεπτα μετά lock=0
    halfCube(shape_3_buffer); // μειώνει το μέγεθος του θησαυρού στο μισό
}
else if (lock2 == 1 && ((clock() - lock_start_time2) * 1000 / CLOCKS_PER_SEC >= lock_duration2)) {
    lock2 = 2; //αν είναι κλειδωμένο και έχουν περάσει 5 δευτερόλεπτα δηλαδή μία μόνο φορά
    //πριν προλάβει να ξεκλειδώσει lock=2
}

if (lock2 != 2) {
    glDrawArrays(GL_TRIANGLES, 0, 36); //εαν lock=2 μην ζωγραφίσεις τον θησαυρό
}
```

Σημείωση: Η συμπεριφορά του flag lock 2 ανάλογα με τις τιμές του είναι

lock2=0: Το flag είναι ξεκλειδωτό. Ο θησαυρός είναι στο κανονικό μέγεθος και μετακινείται.

lock2=1: Ο Χαρακτήρας έπιασε τον θησαυρό αλλά δεν πέρασε το lock\_duration2= 5sec. Ο θησαυρός μειώνεται στο μισό μέγεθος και παραμένει ακίνητος.

lock2=2 : Ο Χαρακτήρας έπιασε τον θησαυρό και πέρασε το lock\_duration2 = 5sec. Ο θησαυρός εξαφανίζεται.

**5ο ερώτημα:** Σε αυτό το σημείο καλούμαστε να υλοποιήσουμε κάποιες λειτουργίες της κάμερας , κάποιες από αυτές ήταν στην προηγούμενη άσκηση και μερικές καινούργιες. Στην προηγούμενη άσκηση η κάμερα δεν δούλευε απολύτως σωστά και είχε προβλήματα, κάτι που φτιάξαμε στο 3ο μέρος. Για να γίνει αυτό έγινε νέος σχεδιασμός της κάμερας από την αρχή. Πρώτα από όλα έχουμε την κίνηση γύρω από τον x άξονα με τα πλήκτρα w,x. Στην main δημιουργήσαμε μία μεταβλητή rotationX η οποία καταχωρεί την γωνία περιστροφής γύρω από τον x άξονα. Η μεταβλητή αυτή θα αλλάζει κατά 1 ή κατά -1 με το πάτημα το w ή του x αντίστοιχα. Το μόνο που μένει είναι να κάνουμε τον μετασχηματισμό που στην περίπτωση μας θα είναι περιστροφή αλλάζοντας μόνο το x με την εντολή :

*Model = glm::rotate(Model, glm::radians(rotationX), glm::vec3(1.0f, 0.0f, 0.0f));*

Όμοια πράττουμε και για τον y άξονα με τον πάτημα των πλήκτρων q και z αλλάζουμε την τιμή μιας 2ης μεταβλητής rotationY κατά -1 ή 1 και στο τέλος κάνουμε πάλι rotate αυτή την φορά με *vec3(0.0f, 1.0f, 0.0f)*; Για την Μεγέθυνση και σμίκρυνση η νέα μεταβλητή που δημιουργήσαμε zoom περιορίζεται στις τιμές τις έτσι ώστε να μην πηγαίνει πίσω από την σκηνή ή πολύ μακριά από αυτήν. Η λειτουργία γίνεται με το +- στο numerical keyboard καθώς και με τα αντίστοιχα τους που βρίσκονται δεξιά του backspace. Τέλος για να γίνει η περιστροφή πρέπει να κάνουμε τον κατάλληλο μετασχηματισμό όπως και πριν, τώρα αλλάζουμε το projection: *glm::mat4 Projection = glm::perspective(glm::radians(zoom), 4.0f / 4.0f, 0.1f, 100.0f);*

Η καινούργια λειτουργία της κάμερας αυτή την φορά θα είναι η μετακίνηση στους άξονες x και y (panning). Δεν είμαστε απόλυτα σίγουροι για το αν το ζητούμενο της άσκησης είναι αυτό που υλοποιήσαμε καθώς συνήθως το panning γίνεται σε κινούμενα αντικείμενα χωρίς την μετακίνηση της κάμερας, ωστόσο υποθέσαμε ότι η λειτουργία που πρέπει να γίνει είναι η μετακίνηση της κάμερας πάνω κάτω δεξιά και αριστερά αντίστοιχα με το πάτημα κάθε πλήκτρου. Για αυτό τον λόγο δημιουργήσαμε πάλι 2 μεταβλητές για panning στο x και y άξονα, αυτή την φορά είναι float καθώς διαπιστώσαμε πως η μετακίνηση κατά μία ολόκληρη

μονάδα ήταν πολύ απότομη. Το μόνο που μένει τώρα είναι ο τελευταίος μετασχηματισμός ο οποίος γίνεται με το `Model = glm::translate(Model, glm::vec3(panX, panY, 0.0f));`. Τέλος, υπολογίζουμε το τελικό μητρώο του μετασχηματισμού το οποίο θα βγαίνει από το γινόμενο των Projection View και Model.

**(Α) Μπόνους:** Στο ερώτημα αυτό πρέπει να κάνουμε το texture του θησαυρού να αλλάζει τυχαία, κάθε φορά που αλλάζει θέση ο θησαυρός. Πρώτα από όλα φορτώνουμε τα 2 νέα αρχεία .jpg με την κλήση των εντολών `data2 = stbi_load("coins2.jpg", &w, &h, &comp, 0);` (όμοια και για το `data3`) μέσα στην main. Στην συνέχεια πρέπει να υλοποιήσουμε μια συνάρτηση `updateTexture(int imageselect)` η οποία θα παίρνει σαν όρισμα ένα νούμερο που θα καθορίζει ποια εικόνα έχει επιλεγεί κάθε στιγμή και θα αλλάζει το texture σε αυτή την εικόνα. Η συνάρτηση αυτή θα καλείται στην main εκεί που γίνεται η μετακίνηση του θησαυρού με όρισμα κάθε φορά έναν τυχαίο αριθμό, με αυτόν τον τρόπο:

```
//μετακίνηση θησαυρού
if (!lock && mapArray[newtresureY][newtresureX] && newtresureY != cubeY && newtresureX != cubeX && !lock2) {
    lock = 1; //κλείδωμα
    lock_start_time = clock(); // Καταγραφή χρόνου έναρξης του κλειδώματος όταν τελειώσει lock=0
    int j;
    for (j = 0; j < 108; j += 3) { //για κάθε 3άδα X,Y,Z
        shape_3_buffer[j] = shape_3_buffer[j] - tresureX + newtresureX; //ενημέρωση των ακμών του θησαυρού
        shape_3_buffer[j + 1] = shape_3_buffer[j + 1] + tresureY - newtresureY; //ενημέρωση ακμών
    }
    tresureX = newtresureX; //αλλαγή θέσης X
    tresureY = newtresureY; //αλλαγή θέσης Y
    newtresureX = rand() % 10;
    newtresureY = rand() % 10;
    int newTextureIndex = rand() % 3; // Επιλογή τυχαίου texture
    updateTexture(newTextureIndex); // Αλλαγή του texture
} else {
    newtresureX = rand() % 10;
    newtresureY = rand() % 10;
}
```

Η Συνάρτηση `updateTexture` φαίνεται παρακάτω :

```
//συνάρτηση αλλαγής του texture
unsigned char* data1;
unsigned char* data2;
unsigned char* data3;
int w;
int h;
int comp;
unsigned int texture;
void updateTexture(int imageSelect) {
    unsigned char* currentData = (imageSelect == 0) ? data1 // imageSelect = 0 χρήση της εικόνας 1
    : (imageSelect == 1) ? data2 // imageSelect = 1 χρήση της εικόνας 2
    : data3; // imageSelect αλλιώς (δηλαδή αν= 1 χρήση) της εικόνας 3

    glDeleteTextures(1, &texture); // Διαγραφή παλιού texture
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, currentData);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}
```

Τα σημεία που δεν εξηγούνται στα σχόλια είναι ίδια με αυτά που υπήρχαν πιο πριν στην main για το 3ο ερώτημα της άσκησης. Στην εργασία μέσα δίνονται και δύο νέα `coins2.jpg` & `coins3.jpg` Τα οποία χρησιμοποιούνται ως διαφορετικά textures. Για να μην έχουμε προβλήματα με τις UV συντεταγμένες φροντίσαμε να έχουν όλα τις ίδιες διαστάσεις.

**(B) Μπόνους:** Για αυτό το ερώτημα προσθέσαμε οπτικά και ηχητικά εφέ για όταν ο χαρακτήρας ακουμπάει τον θησαυρό. Ξεκινώντας από τα ηχητικά εφέ, κάναμε χρήση της βιβλιοθήκης *Winmm.lib* και του header file *windows.h* (η εισαγωγή αυτών φαίνεται πάνω στο πρόγραμμα) με αυτά μπορούμε πολύ εύκολα να προσθέσουμε ήχο αν πάμε στο σημείο στην *main* όπου ο χαρακτήρας “πιάνει” τον θησαυρό και προσθέσουμε την εντολή *PlaySound(TEXT("effect.wav"), NULL, SND\_FILENAME | SND\_ASYNC);* στα παραδοτέα δίνεται και ένα αρχείο *effect.wav* το οποίο θα ακούγεται. Η ευκολία αυτή στην υλοποίηση έρχεται με το κόστος ότι μπορεί να τρέξει μόνο στα windows και όχι σε άλλα λειτουργικά συστήματα καθώς η βιβλιοθήκη είναι των windows.

## Πληροφορίες σχετικά με την υλοποίηση

Η άσκηση υλοποιήθηκε σε λειτουργικό σύστημα windows με την χρήση του visual studio και τις βιβλιοθήκες GLEW, GLFW και GLM επιπλέον για τον ήχο έγινε χρήση της *Winmm.lib* και του header file *windows.h* οπότε πιθανόν να μην μπορεί να τρέξει σε άλλα λειτουργικά συστήματα. Το executable που δημιουργείται κλείνει μόνο όταν πατηθεί το πλήκτρο SPACE (και όχι όταν ο χαρακτήρας πιάσει τον θησαυρό) , ο χαρακτήρας μετακινείται με τα IJKL και η κάμερα με τα QZWX+- καθώς και με τα TGHB. Πέρα από την κανονική άσκηση έχουν υλοποιηθεί και τα α,β από τα μπόνους. Τα αρχεία για τον πηγαίο κώδικα , το vertex shader και το fragment shader έχουν παρθεί από την προηγούμενη άσκηση και έχουν μετονομαστεί κατάλληλα για το 3ο μέρος της άσκησης.

## Αξιολόγηση της λειτουργίας της ομάδας

Τα πρώτα 2 ερωτήματα είχαν γίνει από την προηγούμενη άσκηση. Όσο για τα υπόλοιπα ο Ηλιάκης Ηλίας ασχολήθηκε κυρίως με τα textures ενώ ο Χαλαϊδόπουλος Μιλτιάδης με τα υπόλοιπα ζητούμενα της άσκησης. Η αναφορά συντάχθηκε και από τα δύο μέλη της ομάδας.

## Πηγές

- Το υλικό του εργαστηρίου στο e-course
- Χρησιμοποιήθηκε και έτοιμος κώδικας από την A2 άσκηση όπως και κείμενο από το Readme της A2.
- <https://freesound.org> για το ηχητικό εφέ
- <https://stackoverflow.com/questions/8671888/moving-camera-in-opengl> για την μετακίνηση της κάμερας
- <https://stackoverflow.com/questions/1565439/how-to-playsound-in-c-using-windows-api> για την συνάρτηση *playSound*
- [https://stackoverflow.com/questions/23150123/loading-png-with-stb-image-for-opengl-texture-gives-wrong-colors?fbclid=IwZXh0bgNhZW0CMTEAAR0alj6S4SfMyQkWN3FVO-yVol4bxuedrH9lWrYQQeZQekzkjGqzXpJmfw\\_aem\\_kzb-NCxy-EAreHMeFtYung](https://stackoverflow.com/questions/23150123/loading-png-with-stb-image-for-opengl-texture-gives-wrong-colors?fbclid=IwZXh0bgNhZW0CMTEAAR0alj6S4SfMyQkWN3FVO-yVol4bxuedrH9lWrYQQeZQekzkjGqzXpJmfw_aem_kzb-NCxy-EAreHMeFtYung)
- [https://github.com/nothings/stb/blob/master/stb\\_image.h](https://github.com/nothings/stb/blob/master/stb_image.h) για να φορτώσουμε το texture