1η Εργαστηριακή Άσκηση Modern OpenGL

ΜΥΥ702: Γραφικά Υπολογιστών και Συστήματα Αλληλεπίδρασης

Καθηγητής Ιωάννης Φούντος

Φοιτητές: Βασίλης Πούλος (2805), Θοδωρής Δήμας (2682), Στέλιος Ζαππίδης (2971)

Εισαγωγή

Σε αυτή την άσκηση είχαμε ως ζητούμενο τον σχεδιασμό μιας τρισδιάστατης σκηνής, αποτελούμενη από διάφορα βασικά γεωμετρικά μοντέλα, χρησιμοποιώντας το API της μοντέρνας OpenGL. Συγκεκριμένα έγινε χρήση της έκδοσης OpenGL 3.3.
Παρακάτω παρουσιάζονται περιληπτικά τα ζητούμενα της άσκησης.

Ερώτημα 1:

Η υλοποίηση ενός παραθύρου διάστασης 600x600 μέσα στο οποίο η OpenGL θα σχεδίαζε την σκηνή, καθώς και ένας διάφανος κύβος (SC) 100x100x100 στο παγκόσμιο σύστημα συντεταγμένων, ο οποίος οριοθετούσε την σκηνή.

Ερώτημα 2:

Ο χρήστης οποιαδήποτε στιγμή πρέπει να έχει την δυνατότητα δίνοντας ένα input, να ζητάει από το σύστημα την δημιουργία ενός νέου τυχαίο γεωμετρικού σχήματος μεταξύ των εξής απλών κύβος, σφαίρα, προβολή μαζί με κατάλληλο φωτισμό. Τα αντικείμενα αυτά πρέπει να αναπηδούν κάθε φορά που συναντούν κάποια πλευρά του κύβου.

Ερώτημα 3:

Μέσα στο κέντρο του κύβου πρέπει να υπάρχει και μια μεγάλη σφαίρα (SPH) με έντονο κόκκινο χρώμα, την οποία ο χρήστης μπορεί να μετακινεί δίνοντας το κατάλληλο input. Τα στοιχειώδης σχήματα πρέπει να αναπηδούν από την σφαίρα με τον ίδιο τρόπο που αναπηδούν από όποια πλευρά του κύβου συναντούν.
Ερώτημα 4: Για να δούμε την σκηνή από διάφορες γωνίες, έγινε χρήση μιας κάμερας, την οποία μπορεί να κινήσει ο χρήστης δίνοντας πάλι το κατάλληλο input.

Ερώτημα 5:

Στην σφαίρα SPH δίνεται η επιπλέον ιδιότητα να εμφανίζεται ένα texture με την χρήση συγκεκριμένου input από τον χρήστη.

Τι πετύχαμε

1. Καταφέραμε να δημιουργήσουμε τον κύβο όπως ζητούσε η άσκηση με την χρήση κατάλληλων μετασχηματισμών scaling και translating. Το παράθυρο που δημιουργούμε έχει τις κατάλληλες διαστάσεις και λαμβάνει οποιοδήποτε input δίνει ο χρήστης με ένα callback στην μέθοδο key_callback. Το παράθυρο αυτό κλείνει με το κουμπί `ESC`.

2. Με το πάτημα του κουμπιού `spacebar` εμφανίζονται νέα αντικείμενα με τυχαία χαρακτηριστικά μέγεθος, ταχύτητα, χρώμα καθώς και σχήμα. Επίσης, έγινε χρήση προοπτικής προβολής αλλά δεν καταφέραμε να παρέχουμε κατάλληλο φωτισμό.

3. Μέσα στον κύβο εμφανίζεται η σφαίρα με έντονο κόκκινο χρώμα και κέντρο το σημείο (50,50,50), το οποίο ταυτίζεται με το κέντρο του κύβου. Η σφαίρα κινείται με τα βελάκια του πληκτρολογίου καθώς και τα κουμπιά `+`,`-` του numbpad. Στην υλοποίηση που παραδίδουμε δεν υπάρχει η δυνατότητα αναπήδησης αντικειμένων από την σφαίρα.

4. Υλοποιήσαμε μια κάμερα η οποία πραγματοποιεί περιστροφική κίνηση ως προς τους άξονες X και Y που περνάνε από το κέντρο του κύβου, ενώ έχει και την δυνατότητα zoom in, zoom out δηλαδή κίνηση πάνω στον άξονα Z. Ο χρήστης μπορεί να κινήσει την κάμερα με τα κουμπιά `A`, `D`, `S`, `X`, `W`, `E`. Επισημαίνουμε πως η κάμερα δεν κινείται σε σταθερή απόσταση από το κέντρο του κύβου, με άλλα λόγια δεν διαγράφει απόλυτη κυκλική γύρω από τους προαναφερθέντες άξονες.

5. Στην σφαίρα έχουμε δώσει την δυνατότητα να φορτώνεται ένα texture, αρκεί να δώσουμε την κατάλληλη διαδρομή στην μέθοδο `loadTexture`, το οποίο μπορεί να *ενεργοποιηθεί* και να *απενεργοποιηθεί* πατώντας το κουμπί `T`.

Από τα επιπλέον ερωτήματα έχουμε προσθέσει την δυνατότητα να μεταβάλλουμε την ταχύτητα των αντικειμένων με τα κουμπιά `<` και `>`.

Οδηγίες Εγκατάστασης

Μέσα στον φάκελο "External Libs" πρέπει να υπάρχουν τα απαραίτητα αρχεί από τις βιβλιοθήκες GL, GLFW, GLM

Το active solution platform είναι x86

Το target platform είναι win32

Το glew32.dll πρέπει να είναι στο ίσιο directory με το solution.

Shaders

Οι shaders είναι υπεύθυνοι για την επικοινωνία μεταξύ του επεξεργαστή και της κάρτας γραφικών. Γράφοντας μικρά προγράμματα για τον vertex shader και τον fragment shader μπορούμε να καθορίσουμε την μορφή των αντικειμένων της σκηνής καθώς και το χρώμα τους.

Έγγραφο .shader

Στην υλοποίηση μας χρησιμοποιούμε ένα αρχείο source code και για τους δύο shaders. Η απόφαση αυτή πάρθηκε, καθώς εξυπηρετούσε η ομαδοποίηση τους με αυτόν τον τρόπο. Η διαφοροποίηση τους επιτυγχάνεται με την επικεφαλίδα #shader και το είδος του shader (vertex, fragment αντίστοιχα). Περισσότερες λεπτομέρειες ως προς τον τρόπο ανάγνωσης δίνονται στην ενότητα ShaderProgram.

Μέσα στον vertex shader περιέχονται τα εξής:

Αρχικά οι δύο μεταβλητές που περιέχονται μέσα σε όλα τα μοντέλα αντικειμένου

vertexPosition_modelspace: πρόκειται για ένα vector 3 μεταβλητών και περιέχει τις συντεταγμένες του αντικειμένου στο σύστημα συντεταγμένων local (model) space.

vertexUV: πρόκειται για ένα vector 2 μεταβλητών και περιέχει τις συντεταγμένες του texture που είναι συνδεδεμένο με το αντικείμενο.

Το vector vertexPosition χρησιμοποιείται μέσα στον vertex shader, ενώ το vertexUV ωθείται στον fragment shader.

Ο vertex shader επίσης περιέχει τα transformation matrices για κάθε σύστημα συντεταγμένων ως uniforms. Το model matrix ορίζεται πριν από κάθε draw call ενός αντικειμένου και μετατρέπει τα local coordinates στο παγκόσμιο σύστημα συντεταγμένων. Έπειτα, το view matrix το οποίο ανανεώνεται μετά από κάθε αλλαγή της κάμερας, μετατρέπει τις συντεταγμένες από το παγκόσμιο σύστημα στο οπτικό σύστημα συντεταγμένων (που ακριβώς είναι το αντικείμενο ως προς την κάμερα).

Τέλος, το projection matrix (στο οποίο χρησιμοποιούμε προοπτική όψη) εξυπηρετεί 2 σκοπούς. Ο πρώτος είναι να καθορίζει πόσο μακριά και πόσο κοντά στην κάμερα μπορούμε να δούμε και ο δεύτερος είναι τα αντικείμενα όσο απομακρύνονται από την κάμερα να μικραίνουν (αποτέλεσμα της προοπτικής όψης), το οποίο μας δίνει στην αίσθηση ότι κοιτάμε τρισδιάστατα αντικείμενα.

Για την ενεργοποίηση και απενεργοποίηση του texture της σφάιρας χρησιμοποιούμε την παρακάτω εξήσωση. Όταν το `vec4 textureFlag` είναι (0.0f, 0.0f, 0.0f, 0.0f) το αριστερό κομμάτι της εξίσωσης γίνεται μηδέν και ενεργοποιήται το χρώμα ενώ στην αντίθετη περίπτωση ενεργοποιείται το texture.

```
void main() {

    // Output color = color of the texture at the specified UV
    // using textureFlag to set a color if there is there is no texture
    out_color = textureFlag * vec4((texture(myTextureSampler, UV).rgb), 1.0f) +
(1.0 - textureFlag) * vec4( color, transparency);
}
```

Η κλάση ShaderProgram

Η κλάση αυτή είναι υπεύθυνη για την υλοποίηση ενός «διαμεσολαβητή» μεταξύ των shaders και του επεξεργαστή, που χρησιμοποιούμε κυρίως για τον ορισμό των transformation matrices. Για την αρχικοποίηση του προγράμματος διαχείρησης των shaders, περνάμε στον constructor την διαδρομή προς τον χώρο που είναι αποθηκευμένο το αρχείο .shader. Εσωτερικά της κλάσης υπάρχει μια δομή shaderProgramSource (στο header της κλάσης), η οποία αποτελείται από δύο string το VertexShaderSource και το FragmentShaderSource όπου και αποθηκεύουμε τον πηγαίο κώδικα του κάθε shader. Η διαδικασία για την κατασκευή ένος shader program είναι η εξής:

1. Μέσα στην μέθοδο Parse shader γίνεται η ανάγνωση του αρχείου `.shader`. Όταν το πρόγραμμα διαβάζει την επικεφαλίδα `#shader vertex/fragment` γνωρίζει ότι ο ακόλουθος κώδικας έχει να κάνει με τον shader που ορίζει η επικεφαλίδα και σώζει τον κώδικά του στο αντίστοιχο string.

2. Έχοντας λάβει τα sources και των δύο shaders, περνάμε μέσα στην μέθοδο create shaders που είναι υπεύθυνη για την δημιουργία του αντικειμένου shaderProgram. Για να δημιουργήσει όμως το αντικείμενο πρέπει πρώτα να γίνει η μετάφραση των δύο shaders. Αυτό αποτελεί ευθύνη της μεθόδου compileShader.

3. Η μέθοδος `compileShader` λαμβάνει το είδος του shader που πρόκεται να μεταφραστεί (ορισμένες από την openGL ως `GL_VERTEX_SHADER` και `GL_FRAGMENT_SHADER`. Η μέθοδος στην συνέχεια χρησιμοποιεί τις μεθόδους της openGL για την δημιουργία του shader που θέλουμε. Αν υπάρξει κάποιο σφάλμα κατά την μετάφραση η μέθοδος εμφανίζει ένα μήνυμα λάθους. Όταν ολοκληρωθεί η διαδικασία αυτή, η μέθοδος θα επιστρέψει ένα GLuint που αντιπροσωπεύει πλέον τον δημιουργημένο shader.

4. Το βήμα 3 πραγματοποιείται 2 φορές, μια για κάθε shader. Έπειτα η ροή επιστρέφει στην create shader, η οποία δεσμεύει τους δύο shaders με το πρόγραμμά μας και καταστρέφει τους εικονικούς shaders, αποδεσμεύοντας έτσι μνήμη.

5. Στο τέλος αυτής της διαδικασία έχουμε κρατήσει μια μεταβλητή `GLuint`, η οποία περιέχει το ID του προγράμματος με το οποίο έχουμε δεσμεύει τους δύο shaders.

Άλλες λειτουργίες που εκτελεί η κλάση αυτή είναι ο ορισμός των uniforms που περιέχονται μέσα στον vertex shader, 1 για κάθε είδους uniform που χρησιμοποιούμε. Εσωτερικά της κλάσης υπάρχει και η μέθοδος getUniformLocation η οποία επιστρέφει ένα `GLuint` με την τοποθεσία της μεταβλητής του shader που επιθυμούμε να χρησιμοποιήσουμε. Τέλος, η μέθοδος getAttribLocation βοηθάει τον καθορισμό του layout των vertex buffer των αντικειμένων επιστέφοντας την τοποθεσία των δύο μεταβλητών που λαμβάνει ο vertex shader απευθείας από τα αντικείμενα.

Για την υλοποίηση του ShaderProgram έγινε αναφορά στο εξής βίντεο: Shader Abstraction in OpenGL από το κανάλι The Cherno

Φόρτωση .obj

Για όλα τα αντικείμενα στο πρόγραμμα έχουμε παράγει αντίστοιχα .obj αρχεία με την βοήθει του blender και τα φορτώνουμε χρησιμοποιωντας τον parsher `loadOBJ` απο το tutorial που μας δείξατε στο εργαστήριο.

Ερώτημα 1

Αφού δημιουργήσουμε το βασικό παράθυρο `600x600` στην μέθοδο `drawSceneCube` χρησιμοποιούμε τα .obj δεδομένα του κύβου που φορτώσαμε και λαμβάνωντας υπ'οψη τις αρχικές τους διαστάσεις μετασχηματίζουμε τον κυβο ώστε να είναι παράλληλος στους άξονες και εκτείνεται από το (0,0,0) μέχρι το (100, 100, 100) στο σύστημα παγκόσμιων συντεταγμένων.

```
void drawSceneCube(Object& SC_cube, ShaderProgram& shaderProgram)
{
    shaderProgram.setUniform4f("textureFlag", 0.0f, 0.0f, 0.0f, 0.0f);
    shaderProgram.setUniform3f("set_color", SC_cube.color[0], SC_cube.color[1],
SC_cube.color[2]);
    shaderProgram.setUniform1f("transparency", 0.1f);
    SC_cube.bindVAO();
```

```
    shaderProgram.setUniform4fv("model_matrix", 1, GL_FALSE,
glm::value_ptr(SC_cube.modelMatrix));
    glDrawArrays(GL_TRIANGLES, 0, SC_cube.m_vertices.size());
    SC_cube.unbindVAO();

}
```

Η διαφάνεια ορίζεται απο το `uniform transparency` το οποίο θέτουμε στο 0.1. Στο `InitWindow` ενεργοποιούμε τις δυνατότητες αναγνωρισης βάθους και διαφάνειας τις openGl ώστε να φαίνεται σωστα να έχουμε τον σωστό τιτλο και να έχουμε την δυνατότητα καταγραφής των πλήκτρων. Με το `ESC` καλούμε το `glfwSetWindowShouldClose(window, GL_TRUE)`.

```
    glfwSetWindowTitle(window, u8"Συγκρουόμενα");
    >>>>....>>>>

  glEnable(GL_DEPTH_TEST);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glfwSetKeyCallback(window, key_callback);
```

Προσέχουμε να ολα τα υπόλοιπα αντικείμενα να ζωγραφίζονται με την σωστή σειρα δηλαδή πρώτα ολα τα άδιαφανη και στην συνέχει τα διαφανή με σειρά πρωτεραιότητας απο τα πίσω προς τα μπρός. Πρίν να ζωγραφιστούν τα διαφανή αντικείμενα απενεργοποιούμε τον `z-buffer` και στην συνεχεια τον ενεργοποιούμε ξανα για να εχουμε σωστή απόδοση της διαφάνειας.

Ερώτημα 2 + Bonus

Η πληκτρολόγηση του πλήκτρου `Space` εντοπίζεται απο το events της openGL όπως ορίστικε στη `glfwSetKeyCallback` στην μέθοδο `InitWindow`. Η μέθοδος `key_callback` καλεί την `genObject` η οποία διαχειρίζεται τα στοιχειώδη αντικείμενα ή όπως τα ονομάσαμε *spawnables*. Ο τύπος, το χρώμα, η αρχική κατεύθυνση και το μέγεθος των σωματιδίων καθορίζεται τυχάια (pseudorandom) με την χρήση της `rand()`.

```
void genObject() {
    srand(time(NULL));

    int randomType = rand() % 3 + 1;
    Object spawnable(randomType);

    spawnable.randomRGB(0);

    float speed = 0.04f;
    float v_x = randomFloat(0.1, 0.9) * speed;
    float v_y = randomFloat(0.1, 0.9) * speed;
    float v_z = randomFloat(0.1, 0.9) * speed;
    glm::vec3 direction_v = glm::vec3(v_x, v_y, v_z);
```

```
    spawnable.initialDirection = direction_v;

    int size = randomFloat(1, 10); // considering scale is applied in a size 2
object
                                   // eg. a .obj cube has a side of 2
                                   // so a min

    spawnable.modelMatrix = glm::translate(spawnable.modelMatrix,
                          glm::vec3(0.0f + size, 0.0f + size, 0.0f + size));
    spawnable.modelMatrix = glm::scale(spawnable.modelMatrix, glm::vec3(size/2,
size/2, size/2));
    spawnObjects.push_back(spawnable);

    // Debbug output
    std::cout << "$Main :: Generating object of type " << randomType <<
        " VAO ID -> " << spawnable.getVertexArrayID() <<
        " size d = " << size<<"\n";
    std::cout << "$Main :: Object type " << randomType << " ID: " <<
        spawnable.getVertexArrayID() << " speed is (" <<
        v_x << ", " << v_y << ", " << v_z << ")\n";
}
```

Συγκεκριμένα για τον τύπο του σωματιδίου, πρώτα παράγουμε εναν τυχαίο ακέραιο στο διάστημα [1,3] και στην συνέχεια τον χρησιμοποιούμε σαν όρισμα στον constructor της κλάσης Object η οποία με την σειρά της παράγει το αντίστοιχο αντικείμενο όπως αριθμήσαμε στις σταθερές `TYPE_*`.
Για το χρώμα χρησιμοποιείται η μέθδος `randomRGB`, η οποία παράγει 3 αριθμους απο το 0 μέχρι το 255 και στην συνέχεια τους προσαρμώζει σε float στο διάστημα [0,1] διαιρώντας με το 255. Η παράμετρος brightness δεν χρησιμοποιείται στα σωματίδια.

```
void Object::randomRGB(int brightness)
{
    srand(time(NULL));
    float r = (rand() % 255 + brightness) / 255.0f;
    float g = (rand() % 255 + brightness) / 255.0f;
    float b = (rand() % 255 + brightness) / 255.0f;
    std::cout << "$Obj :: RGB value set to: " << r << " " << g << " " << b <<
"\n";
    color = glm::vec3(r, g, b);
}
```

Με αντίστοιχο τρόπο η αρχική κατεύθηνση ορίζεται στο διάστημα [0.1,0.9] και το μέγεθος d στο [1,10] χρησιμοποιώντας την μέθοδο `randomFloat`, στην δεύτερη περίπτωση γίνεται μετατροπή σε ακέραιο. Εδώ προσέχουμε ώστε τα αντικείμενα να εμφανίζονται στο <0, 0, 0> αλλα να μήν βγαίνουν έξω απο τα όρια του κύβου SC, για αυτόν τον λόγο λαμβάνεται υπ όψη το d και γίνεται ο κατάλληλος μετασχηματισμός. Επίσης θέλοντας να προσαρμώσουμε τις αρχικές διαστασεις των .obj αρχειων που φορτωσαμε για κάθε αντικείμενο (d = 2) διαιρούμε τον τυχαίο αριθμό size δια 2.

Η αναπήδηση πάνω στην επιφάνεια του κύβου ανιχέυεται στο `main.cpp` στην μέθοδο `drawScene` χρησιμοποιώντας την μέθοδο της κλάσης Object `detectCollision`.

Για το δευτερο bonus ερώτημα χρησιμοποιούμε την παρακάτω μέθοδο η οποία αυξάνει η μείωνει την αρχικη ταχύτητα των σωματιδίων που βρισκονται εκείνη την στιγμή στο SC αναλογα με το όρισμα `acceleration_value`.

```
void accelerateSpawnables(float acceleration_value)
{
    for (it = spawnObjects.begin(); it != spawnObjects.end(); ++it) {
        it->initialDirection = it->initialDirection * acceleration_value;
    }
    std::cout << "$Main :: acceleration set to " <<
glm::to_string(spawnObjects.begin()->initialDirection) << "\n";
}
```

```
void Object::detectCollision(glm::mat4 shpereMatrix)
{

    glm::vec4 landingPosition = modelMatrix * glm::vec4(initialDirection, 1.0f);
    glm::vec4 scaleVec = modelMatrix * glm::mat4(1.0f) * glm::vec4(1.0f, 0.0f,
0.0f, 0.0f);
    int scale = scaleVec.x;
    glm::vec4 shperePosition = shpereMatrix * glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);

    if (landingPosition.y >= 100 - scale) {
        initialDirection = glm::vec3(initialDirection.x, -initialDirection.y,
initialDirection.z);
        return;
    }
    else if (landingPosition.y <= 0 + scale) {
        initialDirection = glm::vec3(initialDirection.x, -initialDirection.y,
initialDirection.z);
        return;
    }
    else if (landingPosition.x >= 100 - scale) {
        initialDirection = glm::vec3(-initialDirection.x, initialDirection.y,
initialDirection.z);
        return;
    }
    else if (landingPosition.x <= 0 + scale) {
        initialDirection = glm::vec3(-initialDirection.x, initialDirection.y,
initialDirection.z);
        return;
    }
    else if (landingPosition.z >= 100 - scale) {
        initialDirection = glm::vec3(initialDirection.x, initialDirection.y, -
initialDirection.z);
        return;
    }
```

```
    else if (landingPosition.z <= 0 + scale) {
        initialDirection = glm::vec3(initialDirection.x, initialDirection.y, -
initialDirection.z);
        return;
    }
}
```

Έδω όταν ενα σωματιδίου ακουμπάει πάνω σε κάποια εσωτερική επιφάνεια του SC τότε θεώρούμε μια επιφάνεια στην οποία είναι παράλληλο το διάνυσμα που ακολουθούσε μέχρι την πρόσκρουση του το σωματιδίου. Επομένως αφού πλέον βρισκόμαστε σε επίπεδο 2 συντεταγμένων αντιστρέφουμε, σωστά σε κάθε περίπτωση, το προσυμο της κατάλληλης συντεταγμένης του τυχαίου αρχικού διανυσματος του σωματιδίου. Έδω λαμβάνουμε ξανά υπ'οψη και τις διαστάσεις του σωματιδίου ωστε να μήν βγαίνει εκτός του SC.

Ερώτημα 3

Για την κίνηση της σφαίρας υπάρχουν δύο παγκόσμιες μεταβλητες στο header file Application.h, η πρώτη είναι το `vec3 sphereControl` η οποία καθορίζει την κατεύθυνση της κίνησης της σφαίρας, και η μεταβλητή `SPH_speed` που ορίζει την ταχύτητα, δηλαδή την απόσταση της κάθε μετατόπισης.

Κάθε φορά που το παράθυρο λαμβάνει ως event το πάτημα ενός κουμπιού από τα βελάκια ή το `+ -` του numbpad ενός πληκτρολογίου, μεταβάλουμε την μεταβλητή sphereControl κατάλληλα, και μεταβάλλουμε το model Matrix της σφαίρας πριν από κάθε draw call της SPH.

Μενού κίνησης

Τα βελάκια του πληκτρολογίου κινούν την σφαίρα στους άξονες x, y με κατεύθυνση ίδια με αυτή που δηλώνουν Το `+` κινεί την σφαίρα προς τα θετικά του άξονα z, το οποίο έχει κατεύθυνση προς τον χρήστη Το `–` κινεί την σφαίρα προς τα αρνητικά του άξονα z, δηλαδή μακριά από τον χρήστη.

Ερώτημα 4

Η δημιουργία και η διαχείριση της κάμερας γίνετε μέσα από την κλάση Camera. Για την αρχικοποίηση της απαιτούνται η αρχική της θέση `position` που εκφράζει την θέση του ματιού καθώς και ο ορισμός του πάνω μέρους `worldUp`. Προκειμένου να μπορούμε να εναλλάσσουμε το οπτικό πεδίο της κάμερα μας ανανεώνουμε συνεχώς την αντίστοιχη τιμή view που είναι υπεύθυνη. Αυτό συμβαίνει μέσα στην κλάση της κάμερας με την μέθοδο `calculateViewMatrix()`. Εκεί η `LookAt()` δέχεται ως εξής τις παραμέτρους της αρχικής θέσης κάμερας, τον "στόχο" προς τον οποίο θα κοιτάει και το πάνω μέρος. Ο "στόχος" μας παραμένει συνέχεια σταθερός και είναι το κέντρο του κύβου (50,50,50) όπως αντίστοιχα και το πάνω μέρος (`worldUp`). Αυτό που εναλλάσσουμε συνεχώς είναι η θέση της κάμερας. Η αρχική θέση της κάμερας είναι στο (50,50,250). Κατά το zoomIn αλλάζουμε την θέση της κάμερας με την βοήθεια μιας σταθεράς που την έχουμε ορίσει έπειτα από δοκιμές προκειμένου να μπορεί να είναι όσο πιο ομαλή η κίνηση αυτή της κάμερας. Αντίστοιχα και για το ZoomOut. Έτσι αλλάζουμε την θέση της κάμερας ώστε να την μεταφέρουμε πιο κοντά και μακριά αντίστοιχα. Όσον αφορά τις περιστροφές που θα κάνει η κάμερα αυτές γίνονται με την βοήθεια του `quaternion`. Σε κάθε μία από τις περιστροφές που έχουμε επιλέγουμε η κάθε περιστροφή να γίνετε γύρω από έναν άξονα (x και y αντίστοιχα). Εκεί επιλέγουμε την θετική και αρνητική γωνία που θα μας επιτρέψει να κάνουμε την δεξιόστροφή και αριστερόστροφη αντίστοιχα περιστροφή. Η επιλογή της γωνίας αντίστοιχα έγινε έπειτα από δοκιμές στην περισσότερη επιθυμητή περιστροφή για εμάς. Κατά την

περιστροφή γύρω από έναν άξονα θεωρώντας ως κέντρο τον κύβο μας η θέση της κάμερας δεν ανήκει σε έναν κύκλο με σταθερή ακτίνα. Συνεπώς υπάρχουν σημεία που η κάμερα μας θα περιστρέφεται όπως επιθυμούμε κανονικά γύρω από τον αντίστοιχο άξονα αλλά θα μεγεθύνεται ή θα μικραίνει το πλάνο πατώντας μόνο το αντίστοιχο πλήκτρο περιστροφής. Με την χρήση των πλήκτρων για ZoomIn `E` και ZoomOut `W` ο χρήστης είναι σε θέση να επιλέξει κάθε φορά ανάλογα την εστίαση η απομάκρυνση αν θέλει να φτιάξει το πλάνο του. Έτσι έχει την δυνατότητα να εναλλάξει τις γωνίες από τις οποίες θα βλέπει τον κύβο και τα αντικείμενα του.

Τα πλήκτρα που επιτρέπουν τις παραπάνω κινήσεις είναι:

`W`: rotate x positive
`S`: rotate x negative
`A`: rotate y positive
`D`: rotate y negative
`E`: zoom in
`X`: zoom out

Πηγή

Ερώτημα 5

Γιά την φόρτωση των κατάλληλων δεδομένων του texture ακολουθήσαμε το tutorial του learnopengl.com το οποίο προτείνει την χρήση του header `stb_image.h`. Συμπεριλάβαμε την μέθοδο `loadTexture()` η οποία χρησιμοποιέι την `stbi_load` και φορτώνει τα δεδομένα της εικόνας στην openGl ωστε να χρησιμοποιηθούν ως texture. Η μέθοδος αυτή καλέιται στο `main.cpp` κατα την αρχικοποίηση των παραμέτρων της σφαίρας στην μέθοδο `initSPH`.

Η πληκτρολόγηση του πλήκτρου 'T' εντοπίζεται απο το events της openGL όπως ορίστικε στη `glfwSetKeyCallback` στην μέθοδο `InitWindow`.Η μέθοδος `key_callback` καλεί την `switchTexture()` στο αντικείμενο `Object SPH_sphere` ώστε να δηλωθεί οτι θέλουμε να ενεργοποιήσουμε το texture της.

```
void Object::switchTexture()
{
    enableTexture = !enableTexture;
    std::cout << "$Obj :: EnableTexture set to: " << enableTexture << "\n";
}
```

Στην συνέχεια χρησιμοποιήσαμε το `uniform transparency`, με τον τρόπο που εξηγήθηκε παραπάνω, και τελικά ενεργοποιούμε το shader που φορτώσαμε στην SPH sphere οταν η μεταβλητή `enableTexture` είναι true διαφορετικά το texture απενεργοποιείται και ενεργοποιείται το χρώμα της σφάιρας.

```
void drawSPH(Object& SPH_sphere, ShaderProgram& shaderProgram)
{
    SPH_sphere.bindVAO();
    shaderProgram.setUniform1f("transparency", 1.0f);
    // TODO: built this if into a function?
    if (SPH_sphere.enableTexture)
```

```
    {
        shaderProgram.setUniform4f("textureFlag", 1.0f, 1.0f, 1.0f, 1.0f);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, SPH_sphere.texture);
    }
    else {
        shaderProgram.setUniform4f("textureFlag", 0.0f, 0.0f, 0.0f, 0.0f);
        shaderProgram.setUniform3f("set_color", SPH_sphere.color[0],
SPH_sphere.color[1], SPH_sphere.color[2];
    }

    SPH_sphere.moveObject(sphereControl);
    shaderProgram.setUniform4fv("model_matrix", 1, GL_FALSE,
glm::value_ptr(SPH_sphere.modelMatrix));
    glDrawArrays(GL_TRIANGLES, 0, SPH_sphere.m_vertices.size());
    glBindTexture(GL_TEXTURE_2D, 0);
    SPH_sphere.unbindVAO();
}
```

Μπορείτε και εσείς να χρησιμοποιήσετε τα textures που δοκιμάσαμε δίνωτας η δικά σας ορίζωντας το σωστο path στην `loadTexture(PATH)` στην `initSPH`. Τα textures μας βρίσκονται στον φάκελο `\res\textures`.