ΓΡΑΦΙΚΗ ΜΕ ΥΠΟΛΟΓΙΣΤΕΣ

ΕΡΓΑΣΙΑ ΙΙΙ - ΑΝΑΦΟΡΑ

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2023

ΑΝΑΣΤΑΣΙΟΣ ΣΤΕΡΓΙΟΥ | 10079
ΤΗΜΜΥ ΑΠΘ | anasster@ece.auth.gr

ΕΙΣΑΓΩΓΗ

Το παρόν έγγραφο αποτελεί μία αναφορά στο τρίτο μέρος του μαθήματος *Γραφική με Υπολογιστές* με θέμα *«Θέαση»*. Αρχικά, θα παρουσιαστεί πολύ σύντομα ο στόχος της εργασίας αυτής, και η λειτουργία του παραδοτέου προγράμματος. Στη συνέχεια θα παρουσιαστεί αναλυτικά η λειτουργία των συναρτήσεων που χρησιμοποιήθηκαν στο πρόγραμμα αυτό και τέλος θα παρουσιαστούν και θα σχολιαστούν τα αποτελέσματα.

ΛΕΙΤΟΥΡΓΙΑ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ

Σκοπός της εργασίας αυτής είναι ο χρωματισμός ενός τρισδιάστατου αντικειμένου με βάση το πλήρες μοντέλο φωτισμού *Phong*. Ο χρωματισμός θα γίνει με δύο μεθόδους: τη μέθοδο *Gouraud* με χρήση παρεμβολής μονάχα στα χρώματα των κορυφών των τριγώνων του αντικειμένου, καθώς και τη μέθοδο *Phong*, με χρήση παρεμβολής τόσο στα χρώματα όσο και στα κανονικά διανύσματα των κορυφών του αντικειμένου. Αρχικά, δημιουργούνται τα αντικείμενα που αντιπροσωπεύουν την υφή του υλικού και τις πηγές φωτός που φωτίζουν το αντικείμενο. Στη συνέχεια, δημιουργείται η συνάρτηση που υπολογίζει το φωτισμό ενός σημείου του υλικού αυτού μέσω του μοντέλου φωτισμού *Phong*, καθώς και η συνάρτηση υπολογισμού των κανονικών διανυσμάτων της κάθε κορυφής του αντικειμένου. Παράλληλα, δημιουργούνται και δύο shaders οι οποίοι χρωματίζουν ένα αντικείμενο με τις μεθόδους *Gouraud* και *Phong* αντίστοιχα. Τέλος, υλοποιείται μία συνάρτηση η οποία αφού προβάλλει το αντικείμενο στο raster της κάμερας, το χρωματίζει ο shader με τη μέθοδο που υπαγορεύεται από το χρήστη.

ΑΝΑΛΥΣΗ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ

Στο κομμάτι αυτό, θα περιγραφούν αναλυτικά οι συναρτήσεις και κλάσεις που υλοποιούνται στο πρόγραμμα αυτό. Όπου χρειάζεται θα παρατίθενται μέρη του κώδικα για περαιτέρω επεξηγήσεις.

I. class PhongMaterial

Πρόκειται για την πρώτη από τις δύο κλάσεις που ζητείται να υλοποιηθούν στην εργασία αυτή, και αναπαριστά την υφή του υλικού από το οποίο αποτελείται το αντικείμενο. Ως attributes διαθέτει τους συντελεστές k_a, k_d, k_s για τον φωτισμό λόγω διάχυτου φωτός *(ambient light)*, λόγω διάχυτης ανάκλασης *(diffuse reflection)*, και κατοπτρικής ανάκλασης *(specular reflection)* αντίστοιχα. Επιπλέον συμπεριλαμβάνεται και ο συντελεστής n_{phong} που αφορά τον φωτισμό λόγω κατοπτρικής ανάκλασης. Στην κλάση αυτή υλοποιείται η συνάρτηση constructor ενός αντικειμένου τύπου `PhongMaterial`:

```
def __init__(self, ka, kd, ks, n):
        # Constructor function
        self._validate_data_types(ka, kd, ks, n) # Data type validation
        self.ka = ka # Ambient lighting coefficient
        self.kd = ks # Diffuse reflection coefficient
        self.ks = ks # Specular reflection coefficient
        self.n = n # Phong coefficient
```

καθώς και η βοηθητική συνάρτηση `_validate_data_types` ώστε να εξασφαλίζεται ότι οι συντελεστές φωτισμού θα είναι τύπου `float` και ο συντελεστής *Phong* τύπου `int`.

```
def _validate_data_types(self, ka, kd, ks, n):
        # Function that checks if all the parameters are of the requred data
type
        if not isinstance(ka, float):
            raise TypeError("ka must be a float")
        if not isinstance(kd, float):
            raise TypeError("kd must be a float")
        if not isinstance(ks, float):
            raise TypeError("ks must be a float")
        if not isinstance(n, int):
            raise TypeError("n must be a positive integer")
```

II. class PointLight

Η δεύερη κλάση που υλοποιείται στο πρόγραμμα αυτό είναι η `PointLight` και περιγράφει τις πηγές φωτός που υπάρχουν στη σκηνή. Ως attributes διαθέτει ένα 3×1 διάνυσμα `pos`, και ένα ακόμη 3×1 διάνυσμα `intensity` που περιγράφουν τη θέση της πηγής και την ένταση του φωτός που εκπέμπει αντίστοιχα. Όπως και στην κλάση `PhongMaterial`, υλοποιείται εσωτερικά ένας constructor:

```
def __init__(self, pos, intensity):
        # Constructor
        # Vector validation
        self._validate_vector(pos, (3,))
        self._validate_vector(intensity, (3,))
        self.pos = pos # Source's coordinates in the WCS
        self.intensity = intensity # Source's RGB color intensity
        # Check that all color coefficients of the source's light are in the
[0, 1] interval
        if not np.all((intensity >= 0) & (intensity <= 1)):
            raise ValueError("Intensity values must be in the [0, 1]
interval")
```

ο οποίος μάλιστα πραγματοποιεί και έλεγχο ότι όλες οι χρωματικές συνιστώσες της πηγής βρίσκονται εντός του κλειστού διαστήματος $[0, 1]$. Επιπλέον, υλοποιείται και η συνάρτηση `_validate_vector` ώστε να πραγματοποιηθεί ο έλεγχος ότι πράγματι τα attributes της κλάσης κατά τη δημιουργία ενός αντικειμένου της είναι 3×1 διανύσματα.

```
def _validate_vector(self, vector, shape):
        # Function that validates that all parameters of the class are vectors
of the required shape
        if vector.shape != shape or not isinstance(vector, np.ndarray):
            raise ValueError(f"Vector shape must be {shape} NDArray")
```

III. function light

Η πρώτη συνάρτηση που υλοποιείται θα υπολογίζει το χρώμα ενός σημείου με βάση το πλήρες μοντέλο φωτισμού *Phong*. Συγκεκριμένα, θα υπολογίζει το χρώμα του σημείου λόγω διάχυτου φωτός, διάχυτης ανάκλασης και κατοπτρικής ανάκλασης και στη συνέχεια θα αθροίζει αυτά τα τρία, και θα τα περιορίσει στο κλειστό διάστημα $[0, 1]$. Εφόσον το υλικό διαθέτει τις ίδιες ιδιότητες παντού, αλλά διαφορετικό χρώμα σε κάθε του σημείο, το συνολικό χρώμα του κάθε σημείου θα δίνεται:

$$I_{tot} = c_{point} \cdot (k_a \cdot I_{amb} + \sum_i k_d \cdot I_\iota \cdot < \hat{N}, \hat{V} > + \sum_i k_s \cdot I_\iota \cdot (< 2\hat{N} \cdot < \hat{N}, \hat{L_\iota} > - \hat{L_\iota}, \hat{V} >)^n$$

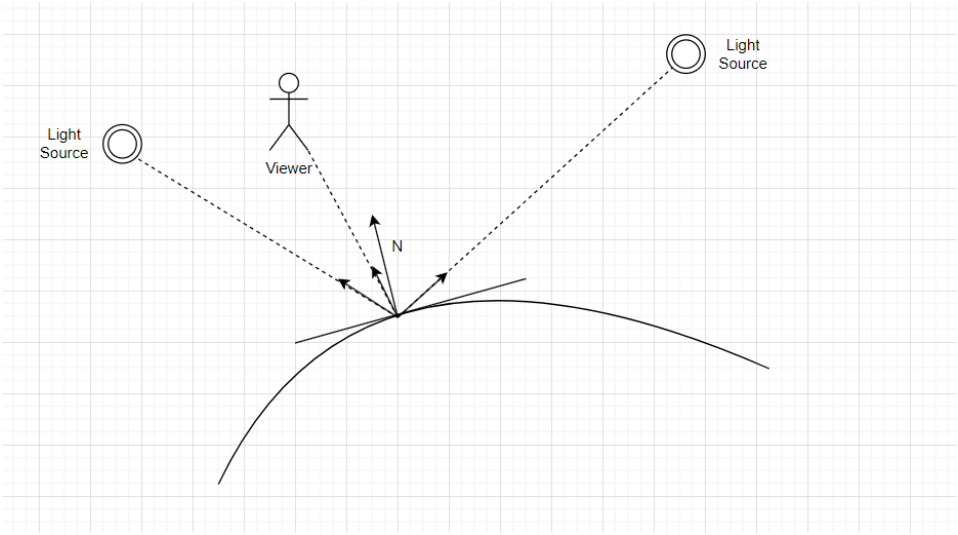
όπου I_ι η ένταση φωτός της i-οστής πηγής, I_{amb} η ένταση διάχυτου φωτός του περιβάλλοντος, $\hat{N}$ το κανονικό μοναδιαίο διάνυσμα του σημείου, $\hat{V}$ το μοναδιαίο διάνυσμα από το σημείο στον παρατηρητή και $\hat{L_\iota}$ το μοναδιαίο διάνυσμα από το σημείο στην i-οστή πηγή. Να τονιστεί σε αυτό το σημείο ότι οποιοσδήποτε πολλαπλασιασμός διανυσμάτων 3×1 (με εξαίρεση τα εσωτερικά γινόμενα) γίνεται element-wise. Η υλοποίηση αυτού του αθροίσματος γίνεται προγραμματιστικά:

```
I = np.zeros(3)
    ambient = mat.ka * color * light_amb # Ambient lighting
    for light in lights:
        # Define vector from point to light source
        l = light.pos - point / np.linalg.norm(light.pos - point)
        l = l/np.linalg.norm(l)

        diffuse = mat.kd * color * light.intensity * np.dot(normal, l) #
Diffuse formula
        specular = mat.ks * color * light.intensity * np.power(np.dot(2 *
normal * np.dot(normal, l) - l, v), mat.n) # Specular formula
        I += diffuse + specular

    I += ambient
    I = np.clip(I, 0, 1) # The final color (i.e. the sum of each illumination
technique) contained in the [0, 1] interval
    return I
```

Το μοντέλο φωτισμού *Phong* οπτικοποιείται παρακάτω:



IV. function calculate_normals

Στη συνέχεια υλοποιείται η συνάρτηση `calculate_normals` μέσω της οποίας υπολογίζονται τα κανονικά διανύσματα για την κάθε κορυφή του αντικειμένου. Από τον πίνακα `verts` μεγέθους $3 \times N_v$ αρχικοποιείται ένας ίδιου σχήματος πίνακας `normals` αρχικοποιημένος με μηδενικά στοιχεία, καθώς και ο μηδενικός μονοδιάστατος πίνακας μήκους N_v όπου θα αποθηκεύεται ο αριθμός των τριγώνων με κοινή κορυφή την i-οστή κορυφή του πίνακα.

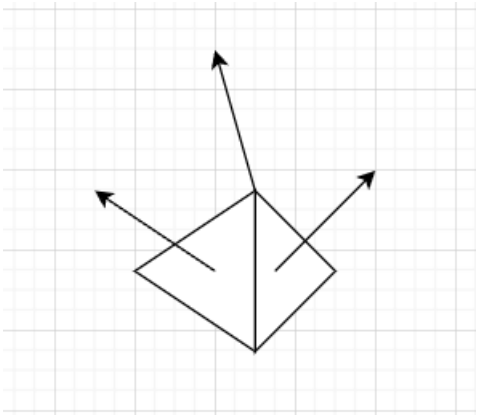
```
# Initialize an empty array for the normal vectors
    normals = np.zeros_like(verts)
    vertex_count = np.zeros(verts.shape[1])
```

Στη συνέχεια, μέσω του $3 \times N_t$ πίνακα τριγώνων `faces`, προσπελαύνεται κάθε τρίγωνο με τη σειρά. Εφόσον είναι γνωστό ότι οι δείκτες της κάθε στήλης του πίνακα παραθέτουν τις κορυφές με δεξιόστροφη φορά, το κανονικό διάνυσμα του κάθε τριγώνου θα προκύπτει από το εξωτερικό διάνυσμα των δύο πλευρών του (με φορά προς τα έξω).

```
for face in faces.T:
        # Save each triangle's index
```

```
        v1_idx, v2_idx, v3_idx = face
        # Calculate the number of triangles this vertex is in
        vertex_count[v1_idx] += 1
        vertex_count[v2_idx] += 1
        vertex_count[v3_idx] += 1
        # Calculate the triangle's edge vectors
        v1 = verts[:, v2_idx] - verts[:, v1_idx]
        v2 = verts[:, v3_idx] - verts[:, v2_idx]
        # Calculate the triangle's normal
        face_normal = np.cross(v1, v2)
```

Στη συνέχεια, τα κανονικά διανύσματα του κάθε τριγώνου αθροίζονται στις κορυφές που συμμετέχουν στο τρίγωνο, και αυτό συμβαίνει για όλα τα τρίγωνα με αποτέλεσμα στο τέλος της προσπέλασης όλων των τριγώνων να έχουμε το άθροισμα των κανονικών διανυσμάτων των τριγώνων με κοινή την i-οστή κορυφή.



```
# A vertex's normal is estimated by the average of the face normals this
vertex belong to
        normals[:, v1_idx] += face_normal
        normals[:, v2_idx] += face_normal
        normals[:, v3_idx] += face_normal
```

Με δεδομένη την παραδοχή ότι το κανονικό διάνυσμα μίας κορυφής είναι ο μέσος όρος των κανονικών των τριγώνων που συμμετέχει η κορυφή αυτή, διαιρούμε αυτό το άθροισμα με τον αριθμό των τριγώνων από τον πίνακα `vertex_count` και κανονικοποιούμε το τελικό αποτέλεσμα, ώστε όλα τα κανονικά διανύσματα να έχουν μοναδιαίο μέτρο.

```
# Divide each vertex's normal by the number of triangles it is a part of, to
take the average
    normals = np.divide(normals, vertex_count)

    return normals / np.linalg.norm(normals, axis=0)
```

V. function shade gouraud

Πρόκειται για τον πρώτο shader του προγράμματος, όπου χρωματίζει τα τρίγωνα του αντικειμένου με τη μέθοδο *Gouraud*, με τα εξής βήματα του αλγορίθμου πλήρωσης αντικειμένου:

- Προσδιορισμός της μορφής του τριγώνου με βάση τον αριθμό οριζοντίων πλευρών του. Έτσι, προκύπτουν τρεις μορφές τριγώνου: Χωρίς οριζόντιες πλευρές, με οριζόντια την κάτω πλευρά του και με οριζόντια την πάν πλευρά του. Επιπλέον, υπάρχει και η ειδική περίπτωση της οριζόντιας γραμμής. Ο διαχωρισμός γίνεται στον κώδικα παρακάτω:

```
# Create two arrays with the sorted indices of x and y coordinates of the
vertices
    sorted_x = np.argsort(verts[0, :])
    sorted_y = np.argsort(verts[1, :])

    # In this part, we check for all possible triangle forms, as well as check
whether the vertices form a straight line
    # Case 1: The triangle has no horizontal sides (default)
    tri_case = 1

    # Now, we check for the rest two cases where two of the triangle's points
are on the same horizontal line, as well as the special case that
    # the vertices form a line. Note that in all cases, the vertices need to
be sorted in a counter-clockwise manner, starting from the bottom vertex

    # Case 2: The first two points (increasing y order) are on the same
horizontal line
    if verts[1, sorted_y[0]] == verts[1, sorted_y[1]]:
        tri_case = 2
        if verts[0, sorted_y[1]] < verts[0, sorted_y[0]]:
            sorted_y = np.array([sorted_y[1], sorted_y[0], sorted_y[2]])

    # Case 3: The last two points (increasing y order) are on the same
horizontal side
    if verts[1, sorted_y[1]] == verts[1, sorted_y[2]]:
        tri_case = 3

    # Case 0: All points belong to the same straight line
    if verts[1, sorted_y[0]] == verts[1, sorted_y[1]] and verts[1,
sorted_y[1]] == verts[1, sorted_y[2]]:
        tri_case = 0
        # Sort vertices and colors by increasing x order
        verts = verts[:, sorted_x]
        colors = colors[:, sorted_x]
        for x in range(verts[0, 0], verts[0, 1]):
            y_line = verts[1, 0] # y coordinate of the straight line
            # Paint the line
```

```
            color = interpolate_vectors(verts[:, 0], verts[:, 1], colors[:,
0], colors[:, 1], x, 1)
            X[y_line, x, :] = color
        for x in range(verts[0, 1], verts[0, 2] + 1):
            y_line = verts[1, 1] # y coordinate of the straight line
            # Paint the line
            color = interpolate_vectors(verts[:, 1], verts[:, 2], colors[:,
1], colors[:, 2], x, 1)
            X[y_line, x, :] = color
        return X

    # We want to sort the vertices in a counterclockwise manner, starting from
the "lowest" lefttmost point (according to its y coordinate)
    if verts[1, sorted_y[0]] != verts[1, sorted_y[2]]:
        # Due to the clockwise sorting method, we want to check whether the
high point or the middle point will have the index 1
        # Create two complex vectors to find the angle between point 0, point
1 and point 2 respectively
        w1 = (verts[0, sorted_y[2]] - verts[0, sorted_y[0]]) + 1j * (verts[1,
sorted_y[2]] - verts[1, sorted_y[0]])
        w2 = (verts[0, sorted_y[1]] - verts[0, sorted_y[0]]) + 1j * (verts[1,
sorted_y[1]] - verts[1, sorted_y[0]])
        if cmath.phase(w1) < cmath.phase(w2):
            sorted_y = np.array([sorted_y[0], sorted_y[2], sorted_y[1]])

    # Since all cases have been checked, we sort the vertices and the colors
    verts = verts[:, sorted_y]
    colors = colors[:, sorted_y]
```

Εδώ πέρα όπως φαίνεται, γίνεται μία ταξιμόμηση σημείων με αντιωρολογιακή φορά, θεωρώντας αρχικό το κάτω αριστερά σημείο του τριγώνου. Έτσι, μέσω της φάσης των μιγαδικών διανυσμάτων που ορίζονται από τις πλευρές του τριγώνου βρίσκουμε ποιο θα είναι το δεύερο σημείο με βάση την αντιωρολογιακή φορά.

- Υπολογισμός ενεργών σημείων και ακμών με βάση τη μορφή του τριγώνου. Τα τρίγωνα με οριζόντιες πλευρές θα έχουν μία ενεργό ακμή ίση με ∞, αφού θα έχει μηδενική κλίση. Ως παράδειγμα, παρουσιάζεται ο αλγόριθμος για τη μορφή χωρίς οριζόντιες πλευρές.

```
if tri_case == 1:
        # No horizontal sides
        effective_points[:, 0] = verts[:, 0]
        effective_points[:, 1] = verts[:, 0]
        # The first effective point is the first vertex of the triangle

        # The inverse slope of the i-th side is the inverse slope between
points i, i + 1
```

```
        slope_inv[0] = (verts[0, 1] - verts[0, 0]) / (verts[1, 1] - verts[1,
0])
        slope_inv[1] = (verts[0, 2] - verts[0, 1]) / (verts[1, 2] - verts[1,
1])
        slope_inv[2] = (verts[0, 0] - verts[0, 2]) / (verts[1, 0] - verts[1,
2])

        # The first two effective sides according to the sorting are 0-1 and
0-2
        effective_sides[0] = slope_inv[2]
        effective_sides[1] = slope_inv[0]
```

- Τέλος, γίνεται πλήρωση του τριγώνου μέσω του αλγορίθμου πλήρωσης, με ενημέρωση των ενεργών σημείων όπως υπαγορεύεται στις σημειώσεις. Στο συγκεκριμένο shader, το χρώμα του κάθε pixel υπολογίζεται μέσω **γραμμικής παρεμβολής των χρωμάτων των κορυφών του τριγώνου**.

VI. function shade phong

Η συνάρτηση αυτή είναι ο δεύτερος shader, μέσω του οποίου το αντικείμενο χρωματίζεται με τη μέθοδο *Phong*. Ο αλγόριθμος που χρησιμοποιείται για τη σκίαση είναι ίδιος με τον *Gouraud* με τη διαφορά ότι τώρα πραγματοποιείται παρεμβολή και στα κανονικά διανύσματα των κορυφών, με αποτέλεσμα το χρώμα του κάθε pixel να υπολογίζεται μέσω της συνάρτησης `light`.

```
elif tri_case == 2:
        # Here there are no special sub-cases. We begin with the effective
points being the two lower vertices
        p1 = np.int64(np.round(np.array([effective_points[0, 0],
effective_points[1, 0]])))
        p2 = np.int64(np.round(np.array([effective_points[0, 1],
effective_points[1, 1]])))
        # Paint the bottom side
        for x in range(p1[0] + 1, p2[0]):
            color_x = interpolate_vectors(p1, p2, colors[:, 0], colors[:, 1],
x, 1)
            norm_x = interpolate_vectors(p1, p2, normals[:, 0], normals[:, 1],
x, 1)
            X[p1[1], x, :] = light(bcoords, norm_x, color_x, cam_pos, mat,
lights, light_amb)
        for y in range(verts[1, 0], verts[1, 2]):
            # Update the effective points' coordinates
            effective_points[:, 0] = np.array([effective_points[0, 0] +
effective_sides[0], effective_points[1, 0] + 1])
            effective_points[:, 1] = np.array([effective_points[0, 1] +
effective_sides[1], effective_points[1, 1] + 1])
```

```
            p1 = np.int64(np.round(effective_points[:, 0]))
            p2 = np.int64(np.round(effective_points[:, 1]))
            # Find the rounded effective points' colors and normals using
interpolation across the y axis, and color the triangle's side points
            color_y1 = interpolate_vectors(verts[:, 0], verts[:, 2], colors[:,
0], colors[:, 2], y + 1, 2)
            color_y2 = interpolate_vectors(verts[:, 1], verts[:, 2], colors[:,
1], colors[:, 2], y + 1, 2)
            norm_y1 = interpolate_vectors(verts[:, 0], verts[:, 2], normals[:,
0], normals[:, 2], y + 1, 2)
            norm_y2 = interpolate_vectors(verts[:, 1], verts[:, 2], normals[:,
1], normals[:, 2], y + 1, 2)
            X[p1[1], p1[0], :] = light(bcoords, norm_y1, color_y1, cam_pos,
mat, lights, light_amb)
            X[p2[1], p2[0], :] = light(bcoords, norm_y2, color_y2, cam_pos,
mat, lights, light_amb)
            # Color the horizontal line between p1, p2
            for x in range(p1[0] + 1, p2[0]):
                color_x = interpolate_vectors(p1, p2, color_y1, color_y2, x,
1)
                norm_x = interpolate_vectors(p1, p2, norm_y1, norm_y2, x, 1)
                X[p1[1], x, :] = light(bcoords, norm_x, color_x, cam_pos, mat,
lights, light_amb)
```

VII. function render object

Τέλος, υλοποιείται η συνάρτηση χρωματισμού του αντικειμένου. Αρχικά, υπολογίζονται τα κανονικά διανύσματα του αντικειμένου, και στη συνέχεια ταξινομούνται τα βάθη με φθίνουσα σειρά. Παράλληλα, προβάλλονται στο πέτασμα της κάμερας τα σημεία του αντικειμένου.

```
k = faces.shape[1]  # Total number of triangles
    l = verts.shape[1]  # Total number of vertices
    p2d, depth = camera_looking_at(focal, eye, lookat, up, verts)
    normals = calculate_normals(verts, faces)
    verts2d = rasterize(p2d, M, N, H, W)

    for i in range(k):
        if not inside_panel(verts2d[:, faces[:, i]], M, N):
            verts2d[:, faces] = np.delete(verts2d[:, faces], faces[:, i])
            depth[faces] = np.delete(depth[faces[:, i]], faces[:, i])

    canvas = np.empty((M, N, 3), dtype=float)
    canvas[:, :] = bg_color
    img = np.ones((M, N, 3)) * bg_color

    # List with the depth of each triangle's vertex
    vdepths = depth[faces]
```

```
    # The depth of each triangle is the mean of its vertices depth
    tdepth = np.mean(vdepths, axis=0)
    # Create a list containing the coordinates of the triangle to be colored
    tcolored = np.empty((2, 3))
    vcolors = np.empty((3, 3))
    norms = np.empty((3, 3))
    depth_indexes = np.argsort(tdepth)[::-1] # Sort the triangles' depths in
decreasing order
    faces = faces[:, depth_indexes]
```

Τέλος, γίνεται η επιλογή shader και χρωματίζεται το αντικείμενο.

```
for i in range(k):
        bcoords = np.mean(verts[:, faces[:, i]], axis=1)
        for j in range(3):
            tcolored[:, j] = verts2d[:, faces[j, i]]
            vcolors[:, j] = vert_colors[:, faces[j, i]]
            norms[:, j] = normals[:, faces[j, i]]
        if shader == 'gouraud':
            img = shade_gouraud(tcolored, norms, vcolors, bcoords, eye, mat,
lights, light_amb, canvas)
        elif shader == 'phong':
            img = shade_phong(tcolored, norms, vcolors, bcoords, eye, mat,
lights, light_amb, canvas)
        else:
            raise ValueError('Shader can only be \'gouraud\' or \'phong\'')

    return img
```

Στο αρχείο `demo.py` το αντικείμενο σκιάζεται και με τα δύο shaders, ξεχωριστά με το κάθε είδος φωτισμού και με τα τρία είδη μαζί.

ΠΑΡΑΤΗΡΗΣΕΙΣ

Στην εργασία αυτή, τα αποτελέσματα δεν φαίνονται σωστά. Συγκεκριμένα, το αντικείμενο δεν εμφανίζεται αρκετά smooth, και οι ακμές των τριγώνων στο αντικείμενο είναι αρκετά ορατές. Θεωρώ ότι το λάθος βρίσκεται στον υπολογισμό των normal, ωστόσο μετά από διάφορες δοκιμές αλγορίθμων vertex normal smoothing δεν βρήκα κάποιο ικανοποιητικό αποτέλεσμα.

ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ