Τμήμα Πληροφορικής
Πανεπιστήμιο Δυτικής Μακεδονίας

Φωτορεαλιστικά γραφικά σε πραγματικό χρόνο με τον αλγόριθμο ray marching χρησιμοποιώντας compute shader

Συγγραφέας:
Κυριάκος Γαβράς

Υπεύθυνος Καθηγητής:
Γεώργιος Σίσιας

1 Αυγούστου 2022

Σύνοψη

Αυτή η διατριβή περιγράφει την υλοποίηση ενός αλγορίθμου 'χύτευσης ακτίνων' με τον οποίο μπορείς να ζωγραφίσεις οτιδήποτε σχήμα στο οποίο μπορείς να ορίσεις την συνάρτηση απόστασής του. Το πρόγραμμα είναι γραμμένο με C++ και χρησιμοποιεί την OpenGL και την GLFW βιβλιοθήκη. Από την στιγμή που η OpenGL ζωγραφίζει μόνο τριγωνα, εφαρμόστηκε μια τεχνική γνωστή ως 'ζωγραφίζοντας κόσμους σε δύο τρίγωνα' η οποία εφαρμόζει δύο τρίγωνα σε όλο το παράθυρο για να δημιουργήσει ένα τετράγωνο και ζωγραφίζει με τον αλγόριθμο μία υφή την οποία εφαρμόζει πάνω στα τρίγωνα αυτά. Ο παράλληλος προγραμματισμός ήταν αναγκαίος εφόσον τρέχει σε πραγματικό χρόνο, για αυτό, το πρόγραμμα χωρίζει την δουλειά σε ομάδες εργασίας ανάλογες με το πλήθος των νημάτων της κάρτας γραφικών.

Abstract

This thesis describes the implementation of a ray casting algorithm in which you can draw any shape, where you can define a distance function for it. The program is written in C++ and uses the OpenGL API and the GLFW library. Since OpenGL can only draw triangles, a technique known as "rendering worlds with two triangles" was applied. This technique draws two triangles that cover the window to create a quad and later on the algorithm draws a texture which is then applied on that quad. Parallel programming was needful since the program runs in real-time, so the program splits the work to different work groups, which they're proportional to the GPU's threads.

Περιεχόμενα

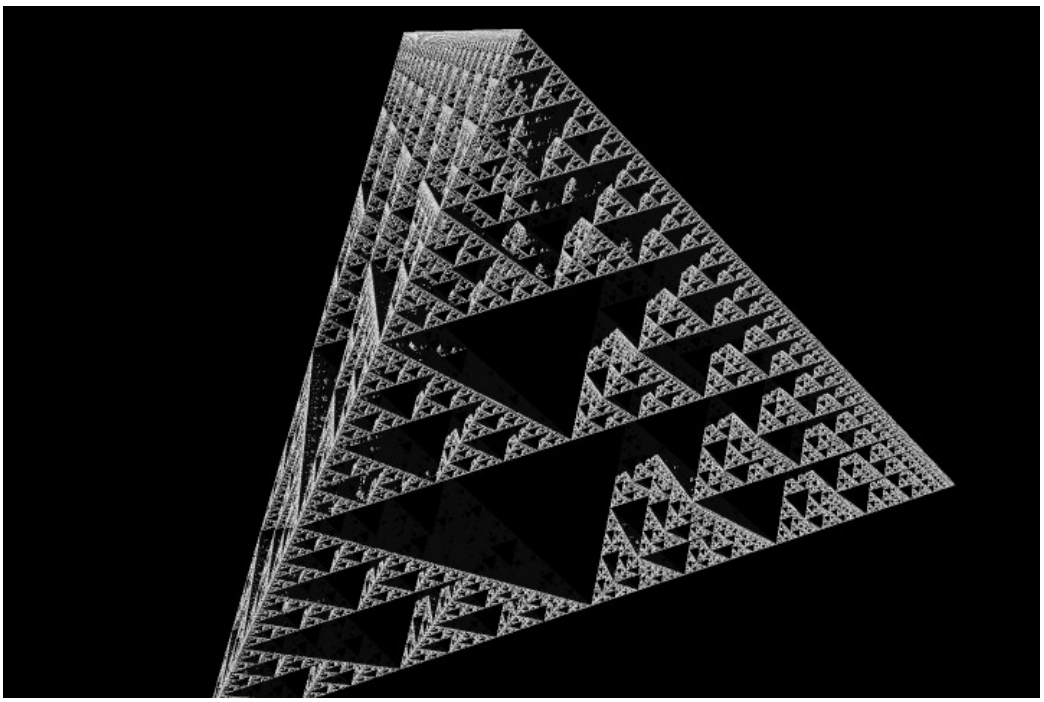
Λίστα Σχημάτων

Κεφάλαιο 1

Εισαγωγή

Αυτή η διατριβή συζητά για την υλοποίηση του αλγορίθμου ray marching μέσα σε έναν compute shader. Ο shader είναι ένα πρόγραμμα που κατά προτίμηση εκτελείται στην κάρτα γραφικών και είναι υπεύθυνος για το τελικό χρώμα ενός αντικειμένου. Υπάρχουν διάφορα είδη shader και το καθένα έχει την δική του λειτουργία, όπως να διαχειρίζονται τις κορυφές των τριγώνων ή το χρώμα αυτών, για φωτισμούς κτλ.

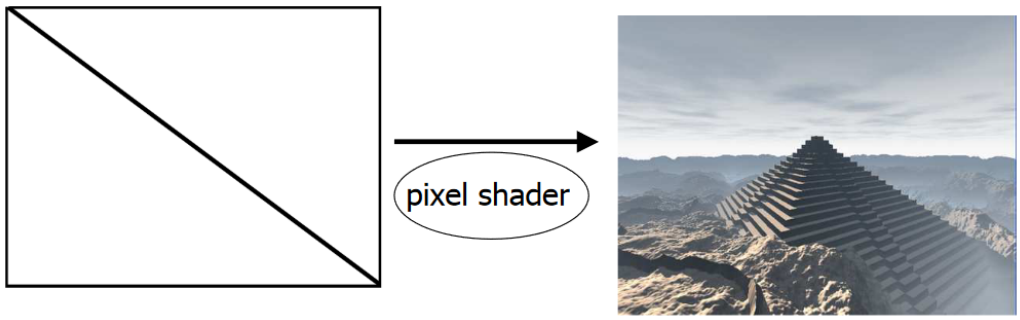
Η τεχνική ray marching προτιμάται να χρησιμοποιείται για την φωτορεαλιστική αναπαράσταση περίπλοκων σχημάτων για τα οποία δεν υπάρχει συνάρτηση σημείου τομής για να αναπαρασταθούν με τον αλγόριθμο ray tracing. Ο αλγόριθμος ray marching μπορεί να χρησιμοποιηθεί για την δημιουργία παιχνιδιών μέσα σε shader **χωρίς** την χρήση πολυγώνων, κινούμενων σχεδίων κ.α. Δείτε την εικόνα 1.1 για ένα παράδειγμα χρήσης του αλγόριθμου ray marching για την αναπαράσταση ενός τρισδιάστατου τριγώνου Sierpinski.



Σχήμα 1.1: Homework 6 for Berkeley CS184 By Kevin Horowitz *(c) copyright 2012, Kevin Horowitz*

1.1 Renderer

Για την υλοποίηση αυτής της εργασίας χρειάστηκε το στήσιμο ενός βασικού renderer. Η OpenGL επιτρέπει την επικοινωνία χρήστη με υπολογιστή και είναι αυτή που δημιουργεί το παράθυρο. Επίσης κάποιες από τις λειτουργίες της είναι να διαβάζει είσοδο από τον χρήστη όπως το ποντίκι ή το πληκτρολόγιο, βασικό κομμάτι για ένα διαδραστικό demo. Στο πρόγραμμα της OpenGL τρέχει η βασική "game loop" όπου καλείται για κάθε καρό που παράγεται. Έπειτα το πρόγραμμα με την game loop χρειάζεται έναν τρόπο για να διαβάζει τα προγράμματα με τους shaders οι οποίοι γράφονται με την γλώσσα προγραμματισμού GLSL. Η μέθοδος που χρησιμοποιείται εδώ είναι να διαβάζει από αρχείο και να τα αποθηκεύει σε string. Θα μπορούσε απευθείας να φτιάχνει το string αλλά δεν είναι καθόλου πρακτικό. Για κάποιες πράξεις γραμμικής άλγεβρας, συναρτήσεις και τύπους δεδομένων χρησιμοποιείται η βιβλιοθήκη GLM της OpenGL. Στην εικόνα 1.2 παρουσιάζεται η προαναφερόμενη τεχνική.



Σχήμα 1.2: Rendering with Two Triangles By Inigo Quilez *(c) copyright August 22 at NVSCENE 08*

Για το αποτέλεσμα της παραπάνω εικόνας στέλνονται οι κορυφές των τριγώνων στον vertex shader από τον buffer και έπειτα αυτά τα τρίγωνα περιμένουν ένα texture στον fragment shader, το οποίο texture θα ζωγραφιστεί στον compute shader, για να εφαρμοστεί πάνω στο τετράγωνο. Αυτό το τέχνασμα επιτρέπει την OpenGL, να ζωγραφίζει pixel ανά pixel αντί για πολύγωνα. Πλέον ο renderer είναι έτοιμος να κάνει render με την χρήση των συναρτήσεων απόστασης (βλ. Κεφάλαιο 3.2). Στην εικόνα 1.3 βλέπετε ένα πραγματικού χρόνου demo με την χρήση συναρτήσεων απόστασης.

Σχήμα 1.3: Happy Jumping By Inigo Quilez *(c) copyright June 2019 on ShaderToy*

1.2 Στήνοντας τον Renderer

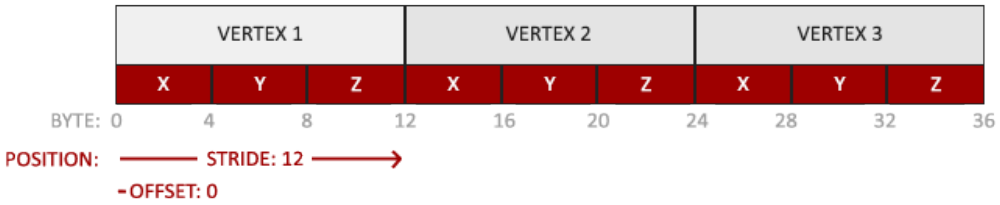
Εφόσον δημιουργηθεί το παράθυρο το επόμενο βήμα είναι να δημιουργηθούν τα δύο τρίγωνα τα οποία θα καλύπτουν όλη την οθόνη. Τα δεδομένα αυτά αποθηκεύονται στον ίδιο buffer μαζί με τις συντεταγμένες της υφής. Οι συντεταγμένες αυτές χρειάζονται για να ξέρει ο renderer που ακριβώς να κολλήσει το texture που θα δημιουργηθεί.

```
float quadVertices[24] = {
        //positions   texture Coords
        -1.0f,  1.0f, 0.0f, 1.0f,
        -1.0f, -1.0f, 0.0f, 0.0f,
         1.0f, -1.0f, 1.0f, 0.0f,

        -1.0f,  1.0f, 0.0f, 1.0f,
         1.0f, -1.0f, 1.0f, 0.0f,
         1.0f,  1.0f, 1.0f, 1.0f
        };
```

Επόμενο βήμα είναι να σταλούν τα δεδομένα στο vertex shader. Υπάρχουν κάποιες συναρτήσεις της OpenGL οι οποίες θα βοηθήσουν τον vertex shader να ξεχωρίσει τα δεδομένα, στην συγκεκριμένη περίπτωση υπάρχουν 2 attributes: θέσεις κορυφών και συντεταγμένες υφών. Γίνεται να δεχτεί μέχρι 16 διαφορετικά attributes.

```
glGenVertexArrays(1, &quadVAO);
glGenBuffers(1, &quadVBO);
glBindVertexArray(quadVAO);
glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices,
    GL_STATIC_DRAW);

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void *)
    0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void *)
    (2 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Η glVertexAttribPointer ξεχωρίζει τα δεδομένα στον buffer. Το πρώτο όρισμα είναι το index που θα χρειαστεί αργότερα στον vertex shader, το πλήθος των στοιχείων (από 1-4), κανονικοποίηση των δεδομένων (είναι ήδη άρα FALSE), το stride και τέλος ο pointer που δείχνει από ποια θέση να ξεκινάει να διαβάζει δεδομένα.



Σχήμα 1.4: Vertex Attribute Pointer από τον Joey de Vries *(c) copyright June 2014*

Στην περίπτωση αυτή τα δεδομένα είναι από 4 float. Το πρώτο στοιχείο που θα σταλεί στον vertex shader είναι οι κορυφές των τριγώνων, άρα ο pointer (τελευταίο όρισμα) είναι στο 0 και τα επόμενα στοιχεία κορυφών βρίσκονται ακριβώς *4 * float* θέσεις μνήμης πέρα. Ισχύει ακριβώς η ίδια λογική και για τις συντεταγμένες των τριγώνων, μόνο που ο pointer ξεκινάει από *2 * float* θέσεις στην μνήμη, γιατί εκεί βρίσκονται τα πρώτα δεδομένα συντεταγμένων και έπειτα με το stride των *4 * float* θα πέφτει μόνο στα δεδομένα συντεταγμένων υφής μέσα στον buffer.

1.2.1 Vertex Shader

Τα δεδομένα του buffer στέλνονται στον vertex shader, ο οποίος θα χειριστεί τις κορυφές των τριγώνων.

```
layout(location = 0) in vec2 aPos;
layout(location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    TexCoords   = aTexCoords;
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

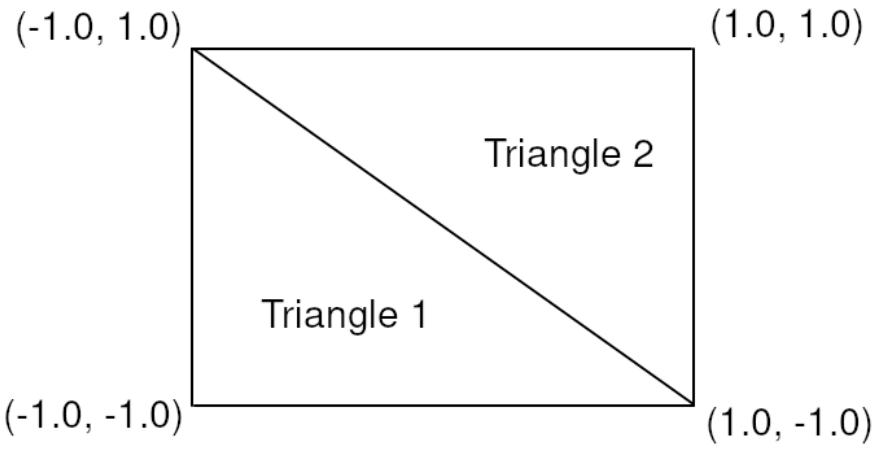
Τα location είναι τα indeces των attribute όπως αναφέρθηκε παραπάνω, στο πρώτο είναι οι κορυφές των τριγώνων και στο δεύτερο οι συντεταγμένες της υφής. Ο vertex shader θα χειριστεί τις κορυφές και θα στείλει τα άλλα δεδομένα στον fragment shader με την εντολή out.

1.2.2 Fragment Shader

Ο fragment shader είναι υπεύθυνος για τον χρωματισμό κάθε fragment των τριγώνων. Με την εντολή in δέχεται τα δεδομένα από τον vertex shader.

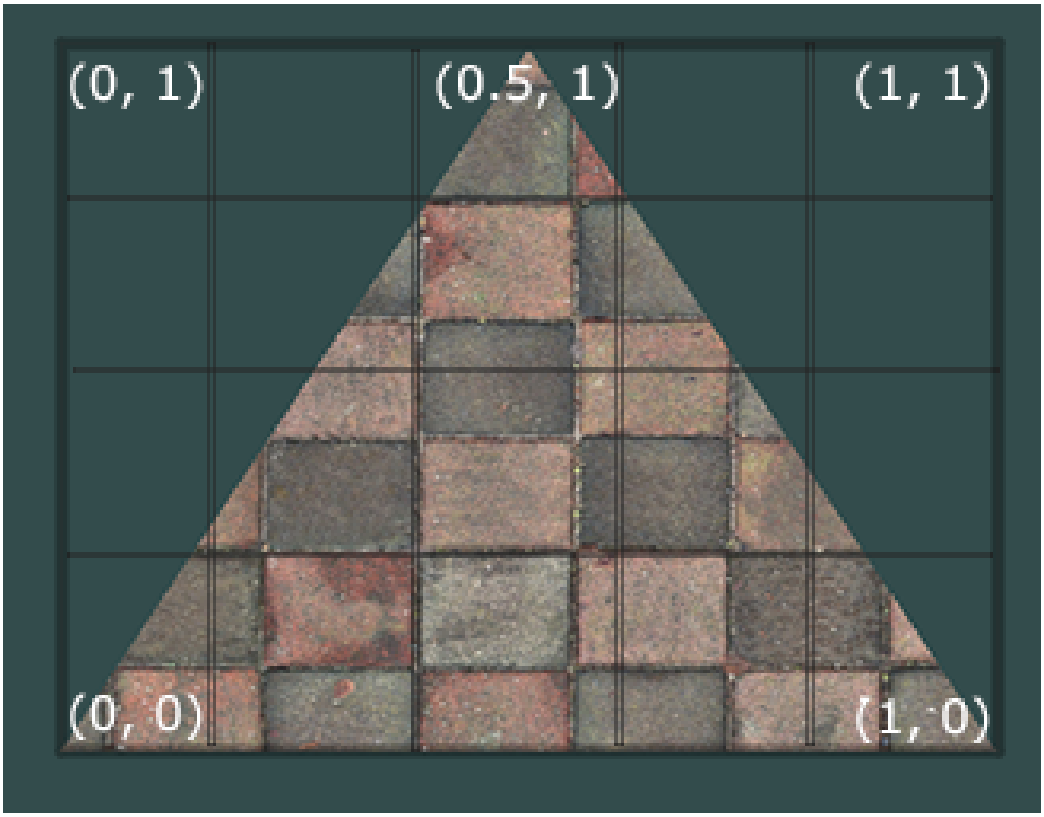
```
#version 330 core

in vec2 TexCoords;
out vec4 color;
uniform sampler2D screenTexture;

void main()
{
    vec3 col     = texture(screenTexture, TexCoords).rgb;
    color = vec4(col, 1.0);
}
```

Η μεταβλητή screenTexture είναι uniform, που σημαίνει ότι έρχεται απευθείας από την OpenGL. Σε αυτήν την μεταβλητή αποθηκεύεται το texture που ζωγραφίστηκε στον compute shader. Τέλος επιστρέφει το texture στο τρίγωνο με τις σωστές συντεταγμένες.



Σχήμα 1.5: Fullscreen Quad

Μέχρι αυτή την στιγμή αυτό είναι το αποτέλεσμα από τον renderer. Δύο τρίγωνα που καλύπτουν όλο το παράθυρο ασχέτου τις διαστάσεις που θα του δοθούν. Οι συντεταγμένες που φαίνονται στην εικόνα 1.5 είναι για τις θέσεις κορυφών. Η OpenGL χρησιμοποιεί διαφορετικό σύστημα συντεταγμένων για τις υφές όπως φαίνεται στην εικόνα 1.6 παρακάτω.



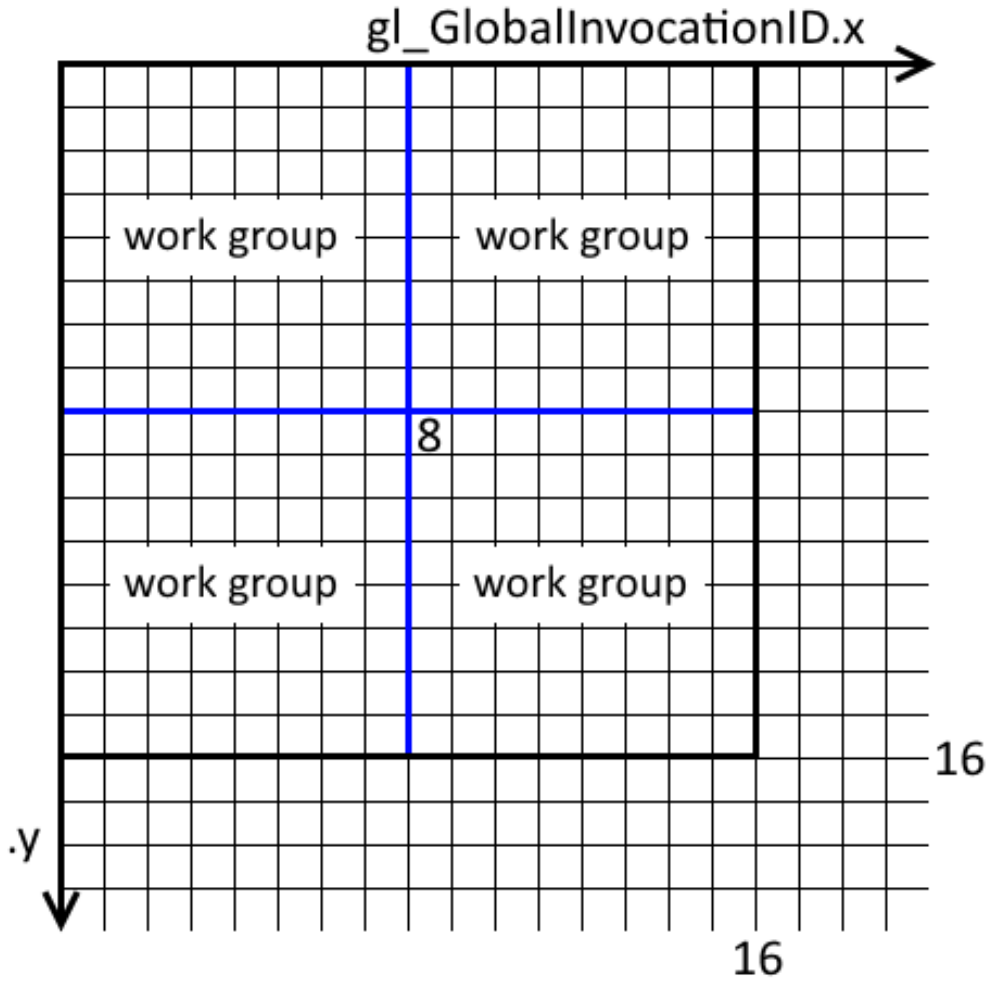
Σχήμα 1.6: Συντεταγμένες υφών στην OpenGL από τον Joey de Vries *(c) copyright June 2014*

1.2.3 Δημιουργία Υφής

Στην δημιουργία της υφής δίνονται οι διαστάσεις, που είναι ίδιες με του παραθύρου και κάποια φίλτρα. Από την στιγμή που η υφή χρωματίζεται από τον αλγόριθμο αργότερα και εφάπτεται σε όλο το παράθυρο δεν θα χρειαστούν τεχνικές όπως MipMaps για καλύτερη απόδοση του προγράμματος. Η συγκεκριμένη υφή χρησιμοποιεί `GL_CLAMP_TO_BORDER` στο Texture Wrapping (κάνει την ίδια δουλειά με το `GL_CLAMP_TO_EDGE` στην περίπτωση αυτή) και για την πληροφορία των pixel `GL_RGBA32F`. Τέλος, το κάνει bind και περιμένει μέχρι να κληθεί ο compute shader.

1.3 Work Groups

Όπως στην OpenCL και CUDA, έτσι και οι compute shaders της OpenGL δουλεύουν σε work groups. Το πρώτο βήμα προτού χωριστούν τα work groups είναι να βρεθούν τα invocations/threads της κάρτας γραφικών. Η OpenGL έχει ειδική συνάρτηση όπου τα επιστρέφει ως τιμή. Έστω ότι έχει 512 νήματα, τότε πρέπει να βρεθεί το ακέραιο μέρος της ρίζας του 512, όπου είναι το 16. Άρα τα work groups θα είναι 16 * 16 (βλ. εικόνα 1.7).



Σχήμα 1.7: Δύο διαστάσεων work groups σε Compute Shader

Τέλος αφού βρεθεί το βέλτιστο πλήθος των work groups καλείται ο compute shader μέσα στο game loop
`glDispatchCompute(SCREEN_WIDTH / workgroups, SCREEN_HEIGHT / workgroups, 1);`. Ουσιαστικά δουλεύουν όλα τα νήματα παράλληλα μέσα στα work groups.

Το πρώτο όρισμα της glDispatchCompute είναι τα work groups στο μήκος, το δεύτερο τα work groups στο ύψος και αντίστοιχα στο βάθος. Το πρόγραμμα αυτό υπολογίζει μία εικόνα άρα τα work groups στο βάθος θα μείνουν 1.

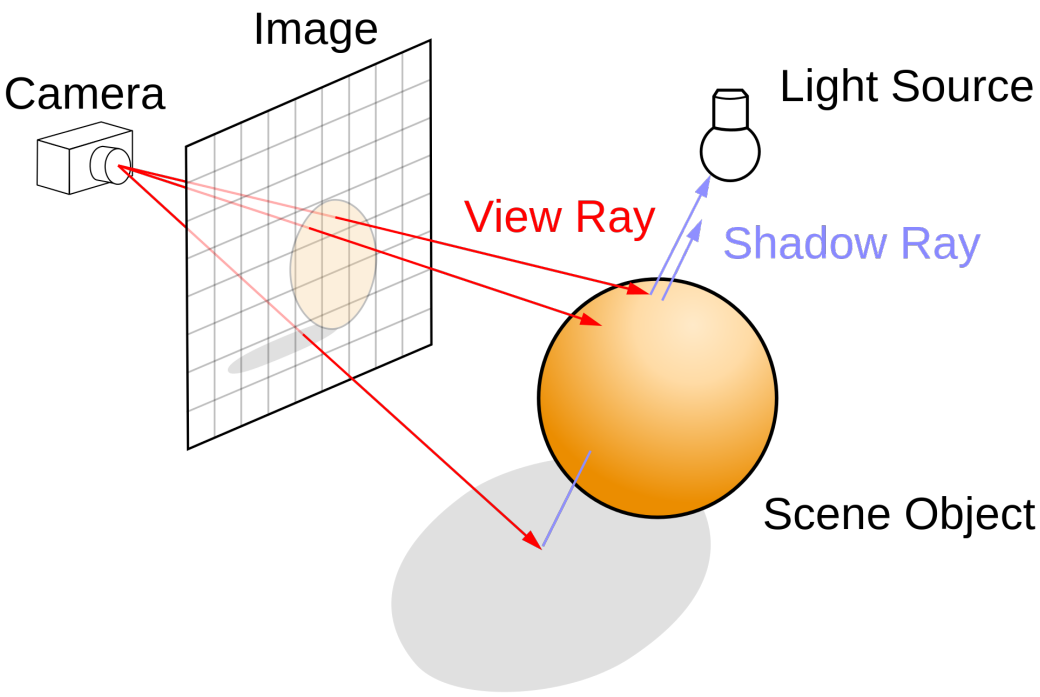
```
uniform uint workgroups;
const uint TILE_W      = workgroups;
const uint TILE_H      = workgroups;
const ivec2 TILE_SIZE = ivec2(TILE_W, TILE_H);

layout(local_size_x = 39, local_size_y = 39) in;
layout(rgba32f, binding = 0) uniform image2D img_output;
```

Ο παραπάνω κώδικας είναι μέσα στον compute shader. Ουσιαστικά αυτό είναι το βήμα της εικόνας 1.7. Ωστόσο τα work groups πρέπει γραφτούν από τον χρήστη διότι η μεταβλητές αυτές δέχονται μόνο θετικές σταθερές και δεν μπορούν να είναι μεταβλητή από uniform.

Κεφάλαιο 2

Εισαγωγή στο Ray Marching

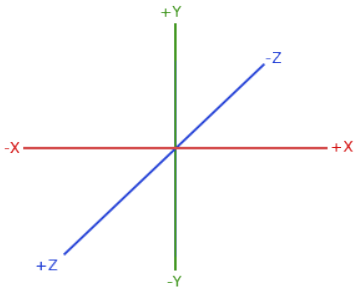
Η τεχνική ray marching μοιάζει πολύ με το ray tracing, ωστόσο έχουν κάποιες διαφορές. Στο ray tracing έχει συναρτήσεις που καθορίζουν το σημείο τομής ενός σχήματος αλλά δεν μπορεί να κάνει render υλικά όπως τα σύννεφα ή άλλα περίπλοκα σχήματα, σε αντίθεση με το ray marching.
Χρησιμοποιώντας τον αλγόριθμο δεν εγγυάται ότι θα εμφανιστούν φωτορεαλιστικά αποτελέσματα, διότι εξαρτάται από τις μαθηματικές φόρμουλες που θα περάσουν φίλτρα στην ακτίνα. Όταν η ακτίνα χτυπήσει ένα αντικείμενο πρέπει να αποφασιστεί τι θα κάνει η ακτίνα, είτε θα επιστρέψει το χρώμα του αντικειμένου που χτύπησε, είτε θα συνεχίσει την διαδρομή της, ανταναμκλώντας από αυτό το αντικείμενο ή ακόμη και να το διαπεράσει (διάθλαση) αν το αντικείμενο είναι διαπερατό από το φως. Να σημειωθεί ότι στην περίπτωση αντανάκλασης, η ακτίνα πρέπει να κρατήσει κάποιο ποσοστό από το χρώμα των προηγουμένων αντικειμένων.



Σχήμα 2.1: Ray tracing από την Wikipedia

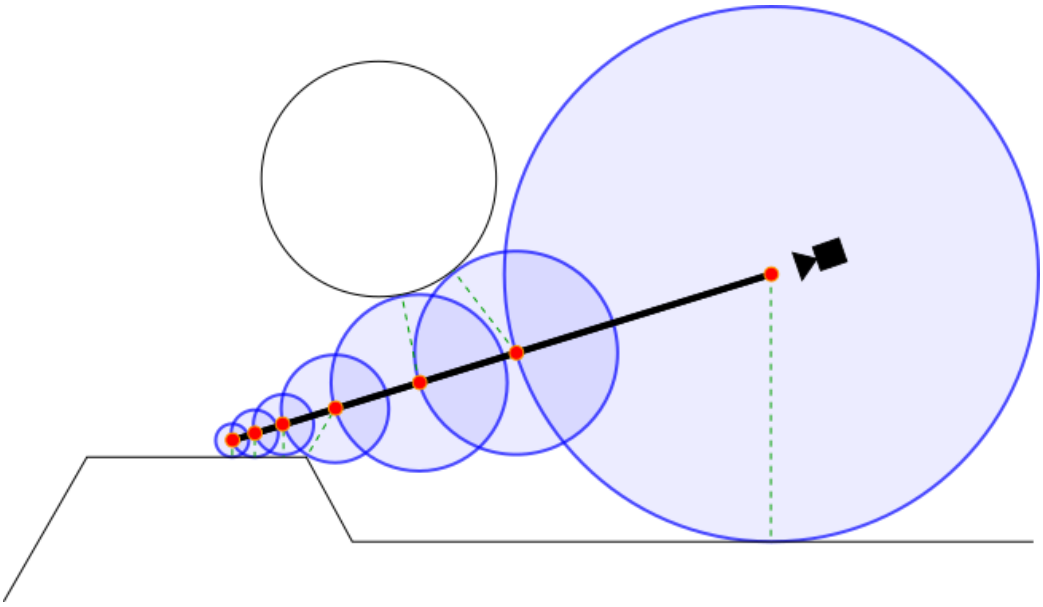
2.1 Ορίζοντας την σκηνή

Βασικό χαρακτηριστικό μιας τρισδιάστατης σκηνής είναι το σύστημα συντεταγμένων, έτσι κάθε αντικείμενο έχει την δική του θέση στον χώρο. Η συγκεκριμένη εργασία χρησιμοποιεί right handed σύστημα συντεταγμένων σαν και αυτό της OpenGL. Μία σκηνή παριστάνεται από αντικείμενα που το καθένα έχει τις δικές του πληροφορίες. Αυτά μπορεί να είναι σχήματα, κάμερα ή φωτισμός. Η κάμερα έχει συντεταγμένες θέσης και διανύσματα για να ξέρει ο υπολογιστής ποιο κομμάτι της σκηνής να ζωγραφίσει, τα αντικείμενα και ο φωτισμός μπορούν να έχουν επιπλέον χαρακτηριστικά που θα αναλυθούν σε μετέπειτα κεφάλαια. Μία ακτίνα θα αναπαραστήσει το χρώμα για το pixel το οποίο ξεκίνησε, το οποίο χρώμα θα επηρεαστεί από τα τρισδιάστατα αντικείμενα της σκηνής και των φωτισμών της.



Σχήμα 2.2: OpenGL coordinate system By Joey de Vries *(c) copyright June 2014*

2.2 Μέθοδος



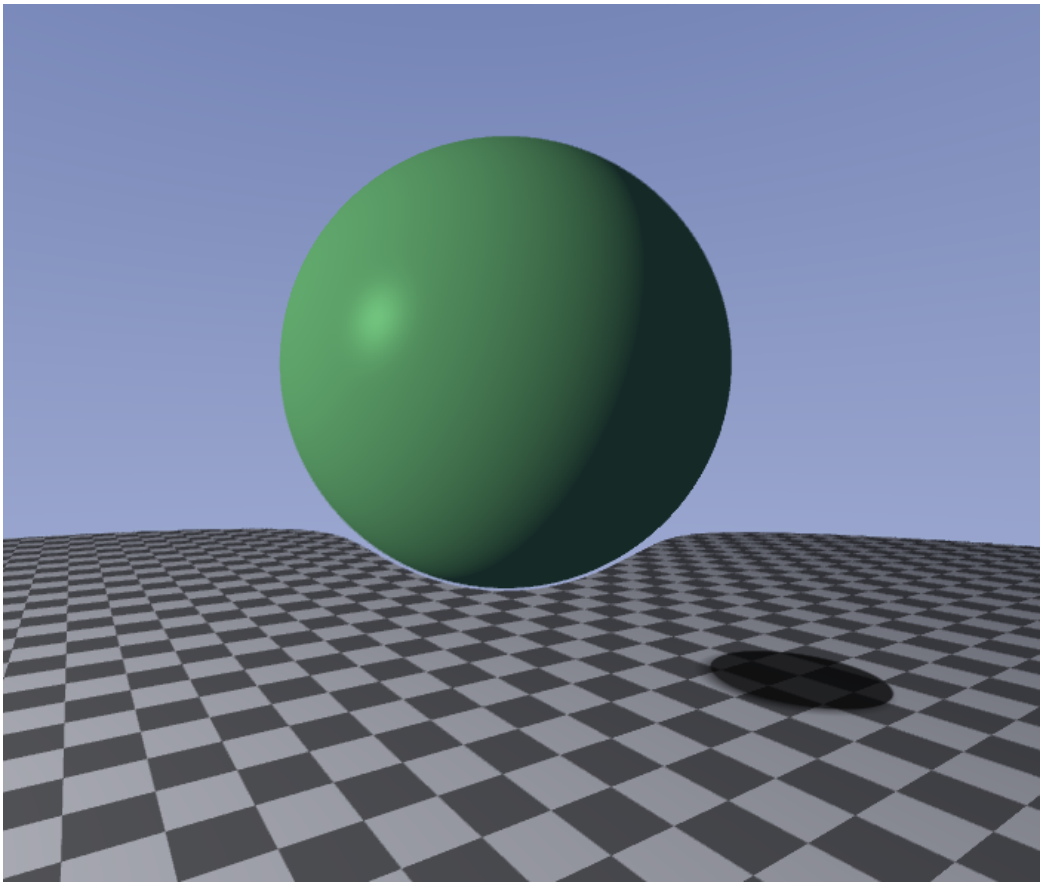
Σχήμα 2.3: Αναπαράσταση του αλγορίθμου ray marching χρησιμοποιόντας συναρτήσεις σημείου επαφής από τον Flafla2 *(c) copyright October 2016*

Στην μέθοδο ray marching η ακτίνα διανύει κάποια απόσταση με βήματα, εξού και η ονομασία. Έστω ότι έχουν οριστεί οι συναρτήσεις απόστασης για όλα τα σχήματα στον χώρο. Ξεκινάει μία ακτίνα από το σημείο της κάμερας με διάνυσμα ίδιο με αυτό της κάμερας. Προτού ξεκινήσει την διαδρομή της ψάχνει το κοντινότερο αντικείμενο συγκρίνοντάς τα.
Υποθετικά όπως φαίνεται στην εικόνα δημιουργείται ένας κύκλος με ακτίνα ίση με την θέση της κάμερας έως το κοντινότερο αντικείμενο. Η ακτίνα θα διανύσει απόσταση όσο αυτή η ακτίνα (διακεκομμένες πράσινες γραμμές), αλλά με διάνυσμα ίδιο με της κάμερας, δηλαδή εκεί που κοιτάζει η κάμερα. Μόλις συμπληρώθηκε το πρώτο βήμα, η διαδικασία θα συνεχιστεί μέχρι η ακτίνα του υποθετικού κύκλου να είναι πολύ μικρή ή ξεπεραστεί ο μέγιστος αριθμός βημάτων.
Η απόσταση αυτή και ο αριθμός βημάτων ορίζεται από τον προγραμματιστή. Άμα ξεπεράσει το όριο τον βημάτων σημαίνει ότι η ακτίνα δεν χτύπησε τίποτα, δηλαδή βρήκε ουρανό. Στην περίπτωση που δεν χτυπήσει κάτι πρέπει να αποφασιστεί τι χρώμα θα επιστρέψει, αργότερα εξηγείται πως χρωματίζεται ο ουρανός. Επίσης άμα ο αριθμός βημάτων είναι πολύ μικρός υπάρχει πιθανότητα να σταματήσει πριν χτυπήσει κάποιο αντικείμενο και να επιστρέψει

το χρώμα του ουρανού αντί του αντικειμένου (βλ. εικόνα 2.4).

Σε ψευδοκώδικα η συνάρτηση που κάνει ray marching:

```
castRay
    for i in step count:
         sample scene
             if within threshold return dist
    return -1
```

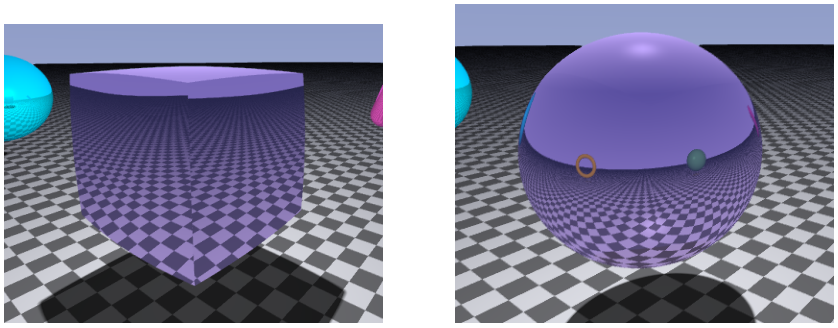


Σχήμα 2.4: Αλλαγή των μέγιστων βημάτων απο 512 στα 100

Στην διατριβή αυτή, η ακτίνα θα σταματήσει όταν θα έχει απόσταση από το αντικείμενο που βρήκε μικρότερη από 0,000001 ή ξεπεραστούν τα 512 βήματα. Σε περίπτωση που η ακτίνα ανταναλαστεί, η αρίθμηση των βημάτων ξαναρχίζει από το 0. Και οι δύο μεταβλητές παίζουν ρόλο στην απόδοση του προγράμματος. Να αναφερθεί ότι οι αριθμοί δεν αναπαριστούν κάποια μονάδα μέτρησης όπως το μέτρο.

2.3 Δυνατότητες

Όπως προαναφέρθηκε πριν, η τεχνική ray marching χρησιμοποιείται για το rendering περίπλοκων αντικειμένων για τα οποία δεν υπάρχουν συναρτήσεις που καθορίζουν τα σημεία τομής τους ώστε να ζωγραφιστούν με ray tracing. Με το ray marching γίνεται να ενωθούν πολλά σχήματα σε ένα ώστε να δημιουργηθεί ένα καινούριο όπως το πλασματάκι στην εικόνα 1.3. Στην εργασία αυτή παρουσιάζεται μία τέτοια τεχνική η οποία ενώνει μία σφαίρα και έναν κύβο. Επειδή είναι τοποθετημένα στο ίδιο σημείο φαίνεται σαν ο κύβος να παίρνει σχήμα σφαίρας και το αντίθετο.



(a) Κύβος (b) Σφαίρα

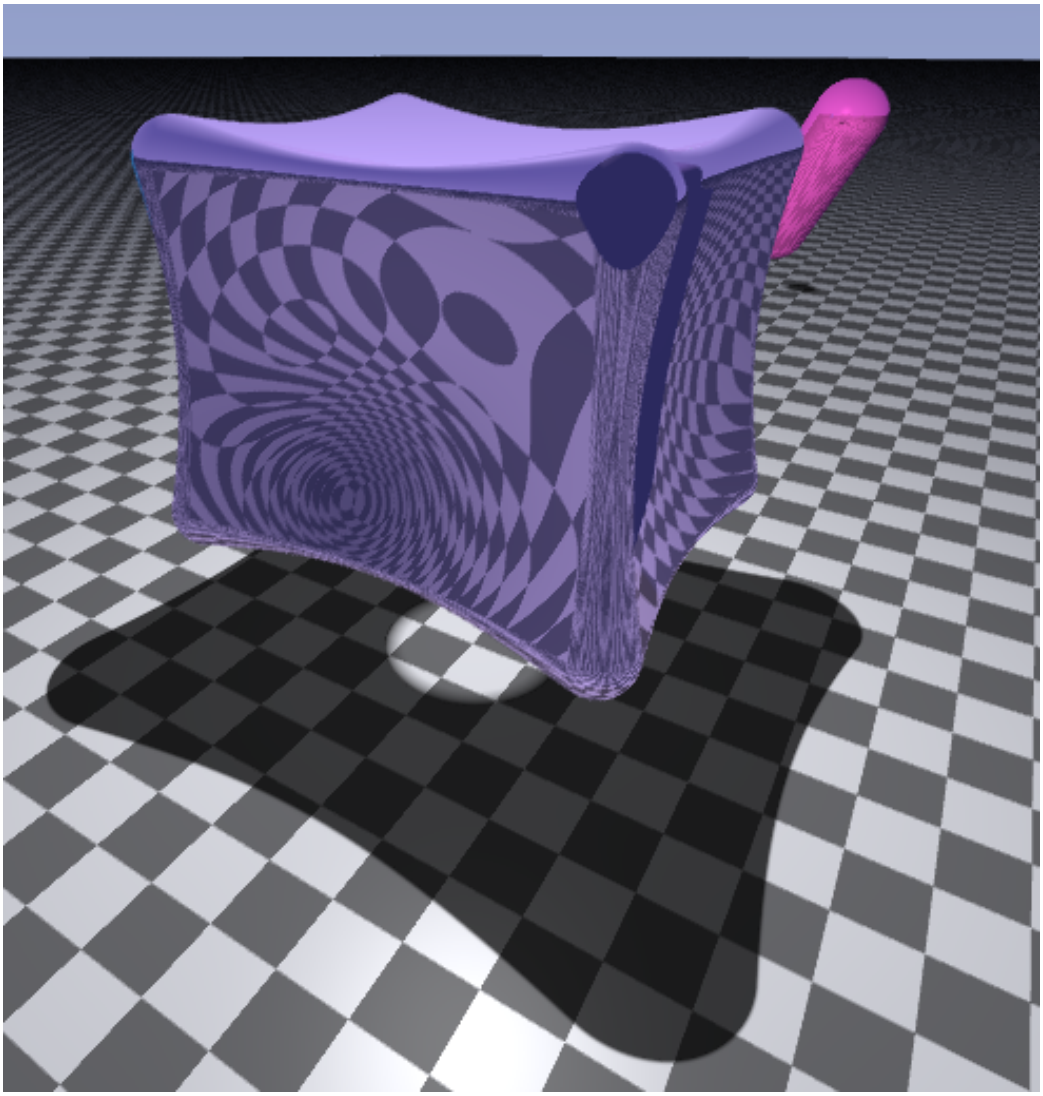
Σχήμα 2.5: Γραμμική παρεμβολή ανάμεσα σε έναν κύβο και μία σφαίρα

Στην εικόνα (a) φαίνεται ότι είναι στο σημείο που αρχίζει να γίνεται ξανά σφαίρα. Χρησιμοποιείται η συνάρτηση mix() της GLSL η οποία κάνει γραμμική παρεμβολή ανάμεσα σε 2 τιμές.

```
t = opU(t, RayHit(mix(Box.hitpoint, Sphere.hitpoint, sin(iTime) * 0.5 + 0.5)
    , vec3(0.4863, 0.3529, 0.702), 4, REFLECTIVE));
```

Μέσα στην mix δίνονται τα σημεία των 2 αντικειμένων και η τρίτη μεταβλητή ορίζει από που έως που θα γίνει η παρεμβολή. Δίνεται σαν τιμή το ημίτονο του χρόνου ο οποίος μετράει σε δευτερόλεπτα και το κανονικοποιεί ώστε να παίρνει τιμές από το 0 έως το 1.

Αφήνοντας τα όρια στις τιμές του ημιτόνου ο κύβος φτάνει σε σημείο να φαίνεται σαν να τον βασανίζανε στην σκάφη. Παρά το γεγονός αυτό, οι ανταναακλάσεις από το πάτωμα στον κύβο είναι ενδιαφέρον.

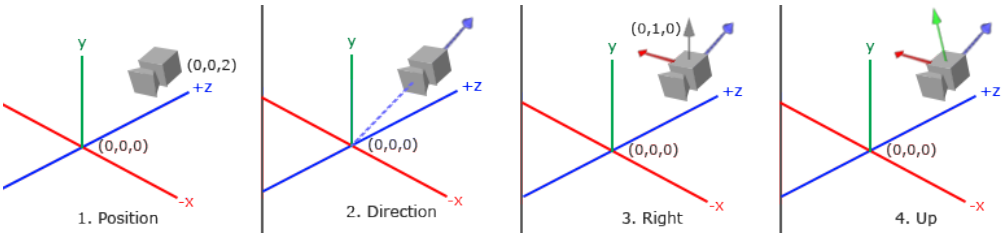


Σχήμα 2.6: Κάνοντας παρεμβολή με τιμή από -1 εώς 1

Κεφάλαιο 3

Υλοποίηση

3.1 Στάσιμη κάμερα

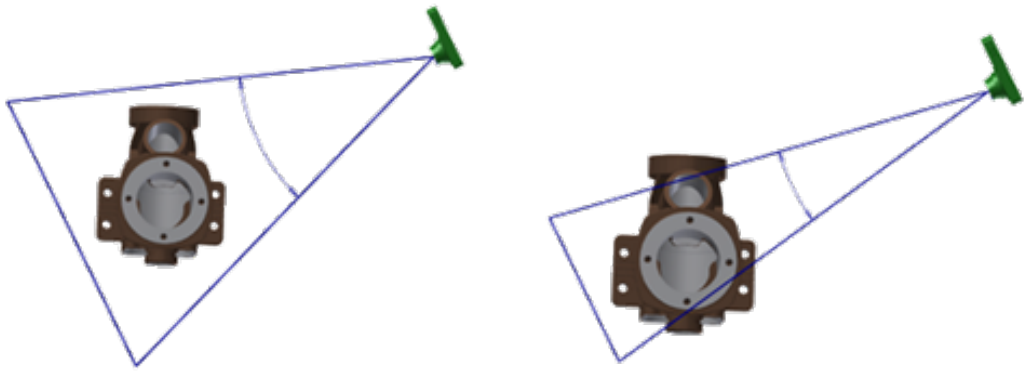


Σχήμα 3.1: Κάμερα στην OpenGL από τον Joey de Vries *(c) copyright June 2014*

Στην συγκεκριμένη εργασία χρησιμοποιείται μια διαφορετική τεχνική για το στήσιμο της κάμερας, αλλά για λόγους απλότητας θα αναφερθεί πως στήνεται μία στάσιμη κάμερα για το πρώτο render. Χρειάζονται 4 πράγματα για να δημιουργηθεί μία κάμερα. Η θέση της στον χώρο, ένα διάνυσμα για την κατεύθυνση στην οποία κοιτάζει, ένα διάνυσμα για τον άξονα x και ένα για τον άξονα y. Εφόσον η κάμερα αυτή δημιουργείται για έναν ray casting αλγόριθμο η υλοποίηση της θα είναι λίγο διαφορετική από αυτή της OpenGL. Στο απόσπασμα κώδικα παρακάτω, υλοποιείται μία συνάρτηση για την δημιουργία μιας κάμερας.

```
vec3 castRay(vec2 uv, vec3 camPos, vec3 camTarget)
{
    vec3 zAxis = normalize(camTarget - camPos);
    vec3 xAxis = normalize(cross(vec3(0.0, 1.0, 0.0), zAxis));
    vec3 yAxis = normalize(cross(zAxis, xAxis));

    float Persp = radians(45);
    vec3 dir = normalize(uv.x * xAxis + uv.y * yAxis + zAxis * Persp);

    return dir;
}
```

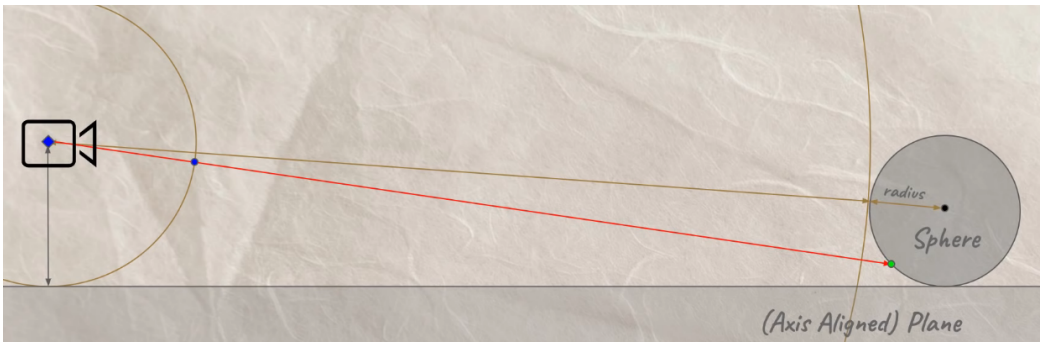
Η uv κρατάει τις τιμές x,y για το pixel από το οποίο θα ξεκινήσει η ακτίνα, η camPos έχει την θέση της κάμερας και η camTarget εκεί που κοιτάζει η κάμερα. Το πρώτο βήμα είναι να βρεθεί το διάνυσμα του z άξονα (άξονας βάθους) αφαιρώντας το σημείο που κοιτάζει η κάμερα με την θέση της. Έπειτα με το διανυσματικό γινόμενο του διανύσματος που κοιτάζει πάνω (στον άξονα y) και το διάνυσμα του z, επιστρέφεται το διάνυσμα του άξονα x. Η ίδια τεχνική εφαρμόζεται και για τον y άξονα εφόσον δεν θα είναι `vec3(0.0, 1.0, 0.0)` σε περίπτωση που η κάμερα κοιτάξει πάνω ή κάτω. Τέλος η μεταβλητή uv αντισταθμίζει την ακτίνα πάνω-κάτω και δεξιά-αριστερά, ανάλογα το pixel στο οποίο γίνεται το ray casting. Ο άξονας z πολλαπλασιάζεται με μία μεταβλητή η οποία δημιουργεί προοπτική, δηλαδή καθορίζει πόσο μακριά θα φαίνονται τα αντικείμενα. Η εικόνα 3.2 δείχνει πως δημιουργείται αυτή η ψευδαίσθηση.



Σχήμα 3.2: Working with Cameras By Brian Ekins *(c) copyright September 2013*

3.2 Συναρτήσεις Απόστασης

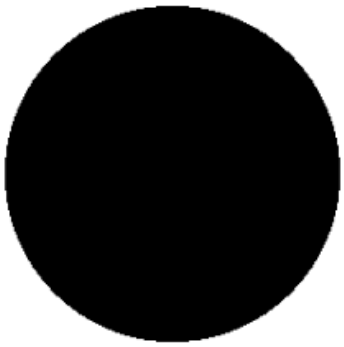
Με τις συναρτήσεις απόστασης ορίζουμε τα σχήματα στον χώρο. Στην πραγματικότητα δεν δημιουργείται κάποιο σχήμα όπως αντικείμενα από προγράμματα όπως το blender, αλλά δίνονται κάποια όρια στον χώρο για να ξέρουν πότε οι ακτίνες πρέπει να σταματήσουν.



Σχήμα 3.3: Youtube video Ray Marching for Dummies! από το κανάλι The Art of Code *(c) copyright December 2018*

```
float sdSphere(vec3 p, float r)
{
    return length(p) - r;
}
```

Αυτή είναι η συνάρτηση απόστασης για μία σφαίρα. Η λογική για την κατασκευή της συνάρτησης έχει ως εξής: Έχοντας την απόσταση από την κάμερα έως το κέντρο της σφαίρας, αφαιρώντας την ακτίνα της σφαίρας, η ακτίνα της κάμερας θα σταματάει πάντα στην επιφάνεια την σφαίρας. Η συνάρτηση παίρνει δύο ορίσματα, το πρώτο είναι αποτέλεσμα αφαίρεσης της θέσης της ακτίνας με το κέντρο της σφαίρας. Καλείται για κάθε βήμα της ακτίνας άρα το p θα διαφέρει κάθε φορά που καλείται η συνάρτηση. Το δεύτερο όρισμα είναι η ακτίνα της σφαίρας. Άρα στην πραγματικότητα δεν υπάρχει καμία σφαίρα, απλώς η ακτίνα ξέρει πότε να επιστρέψει διαφορετικό χρώμα για να ζωγραφίσει μία σφαίρα, η ίδια λογική ακολουθείτε για όλα τα σχήματα.



Σχήμα 3.4: Πρώτο render με συνάρτηση απόστασης

Στην παραπάνω εικόνα φαίνεται το αποτέλεσμα με μία συνάρτηση απόστασης. Μπορεί να πει κανείς ότι δεν είναι σφαίρα αλλά κύκλος ή τουλάχιστον μοιάζει περισσότερο με έναν, παρά μία σφαίρα. Αυτό που θα δώσει την σφαίρα την ιδέα ότι είναι τριών διαστάσεων είναι ο χρωματισμός και ο φωτισμός αυτής. Στο κεφάλαιο 3.5 εξηγείται πως δίνεται η αίσθηση των τριών διαστάσεων χρησιμοποιώντας το normal μιας επιφάνειας.

3.3 Χρωματισμός Ουρανού

Ο χρωματισμός του ουρανού γίνεται σε μία σειρά κώδικα. Χρησιμοποιεί ένα χρώμα και έπειτα το αφαιρεί με μία τιμή, η οποία τιμή που χρησιμοποιείται είναι η θέση του pixel στον άξονα y.

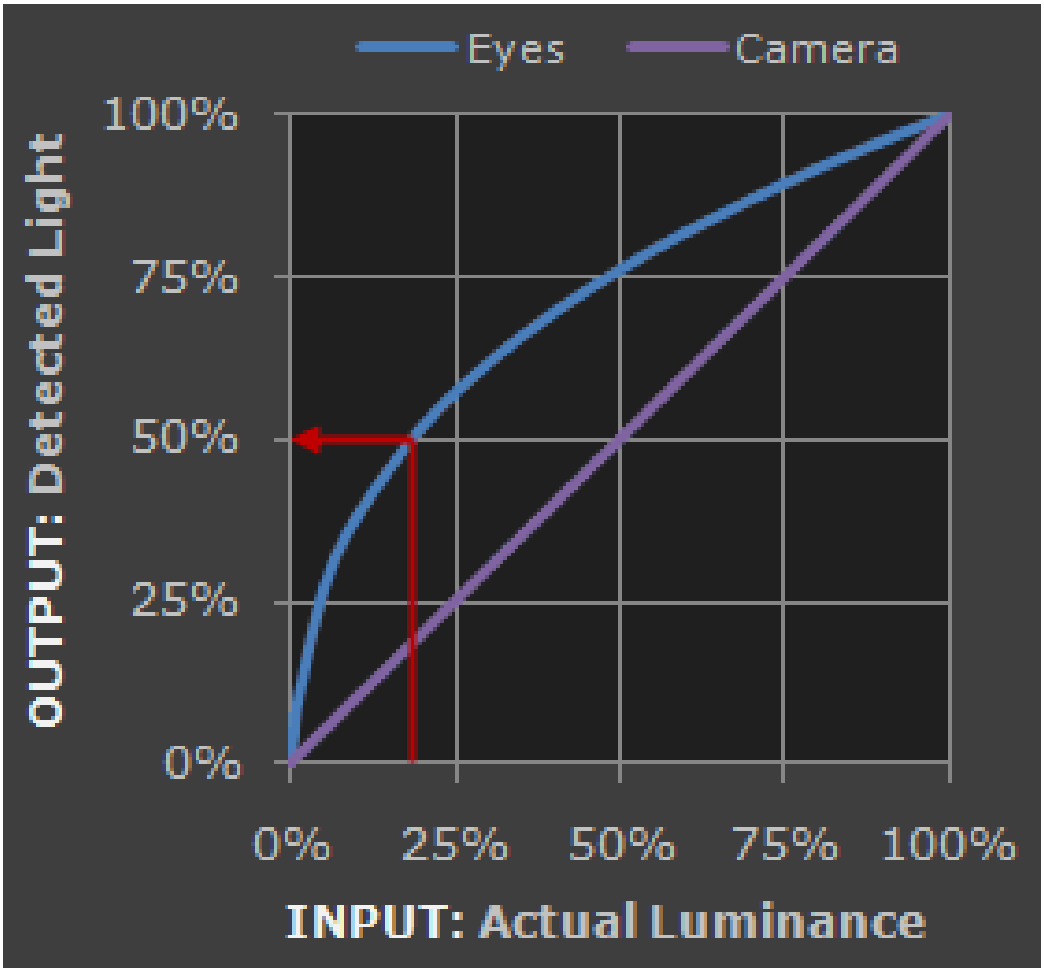
```
vec3 color = vec3(0.30, 0.36, 0.60) - (rayDir.y * 0.2);
```

Το y πολλαπλασιάζεται με μία μικρή τιμή ώστε να μην δίνει μαύρα όταν κοιτάει η κάμερα πάνω. Μέσα στον κώδικα ο χρωματισμός του pixel θα γίνει έτσι μόνο όταν η συνάρτηση που κάνει ray marching επιστρέψει -1, δηλαδή δεν χτύπησε κανένα αντικείμενο.

Σχήμα 3.5: Χρωματισμός Ουρανού.

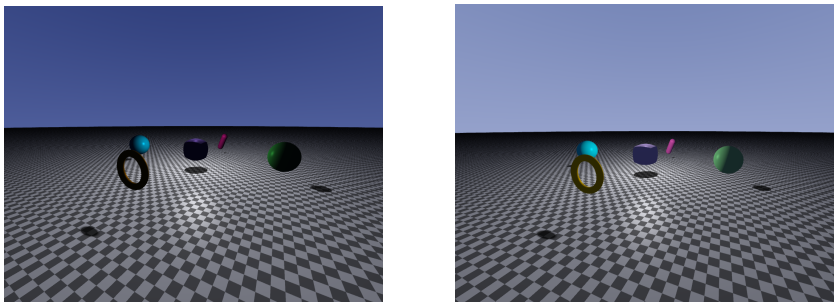
3.4 Gamma Correction

Το ανθρώπινο μάτι δέχεται διαφορετικά τα φωτόνια από ότι μία κάμερα, δεν δέχεται την φωτεινότητα με γραμμικό τρόπο όπως μία κάμερα. Για παράδειγμα, ανάβοντας μία δεύτερη λάμπα μέσα σε έναν χώρο, ο ανθρώπινος εγκέφαλος δεν αντιλαμβάνεται διπλάσια φωτεινότητα. Όπως έχοντας 100 λάμπες αναμμένες, ανάβοντας μία ακόμα δεν θα αποτελέσει αστητή διαφορά. Σε αντίθεση με τις κάμερες, οι κάμερες δέχονται φωτόνια σε γραμμικό τρόπο, δηλαδή έχοντας 2 λάμπες αντί για 1, θα αντιληφθεί και διπλάσια φωτόνια.



Σχήμα 3.6: Gamma Correction από το Cambridge in Colour *(c) copyright 2005-2020*

Επειδή και οι οθόνες δεν δουλεύουν σε γραμμικό χώρο, πρέπει να μετατραπεί το χρώμα σε διαφορετική φωτεινότητα. Επειδή όλες οι οθόνες είναι διαφορετικές, σε κάθε μία το gamma correction πρέπει να είναι διαφορετικό, ωστόσο μια κοινή υπόθεση είναι είναι η παραπάνω μπλε καμπύλη (εικόνα 3.6), που βγαίνει υψώνοντας το χρώμα στην δύναμη 1/2.2 ή 0.4545, `color = pow(color, vec3(0.4545));`.

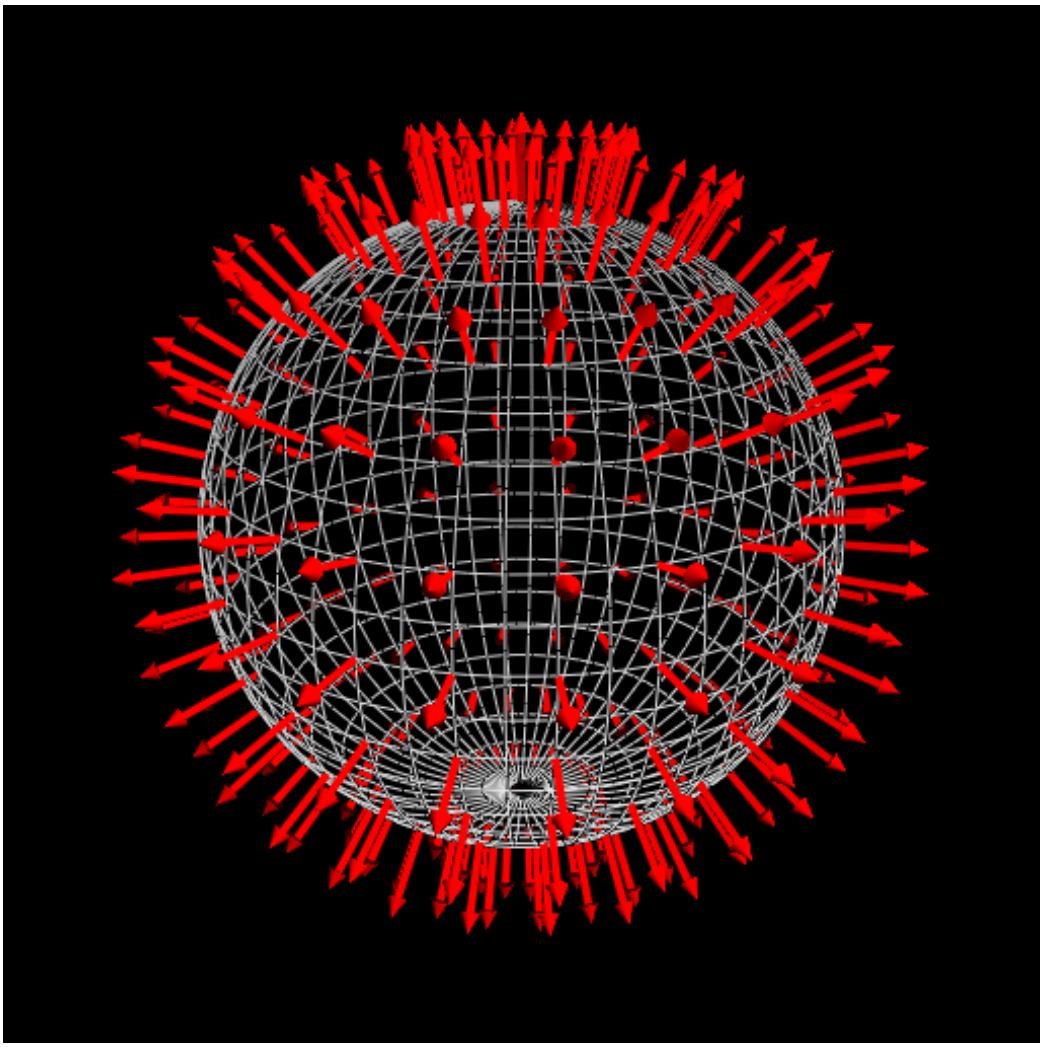


(a) Χωρίς Gamma Correction (b) Με Gamma Correction

Σχήμα 3.7: Διαφορές με και χωρίς Gamma Correction

Από ότι φαίνεται οι διαφορές είναι εμφανείς και δεν πρέπει σε καμία περίπτωση να παραληφθεί το gamma correction.

3.5 Κάθετο Διάνυσμα στην Επιφάνεια

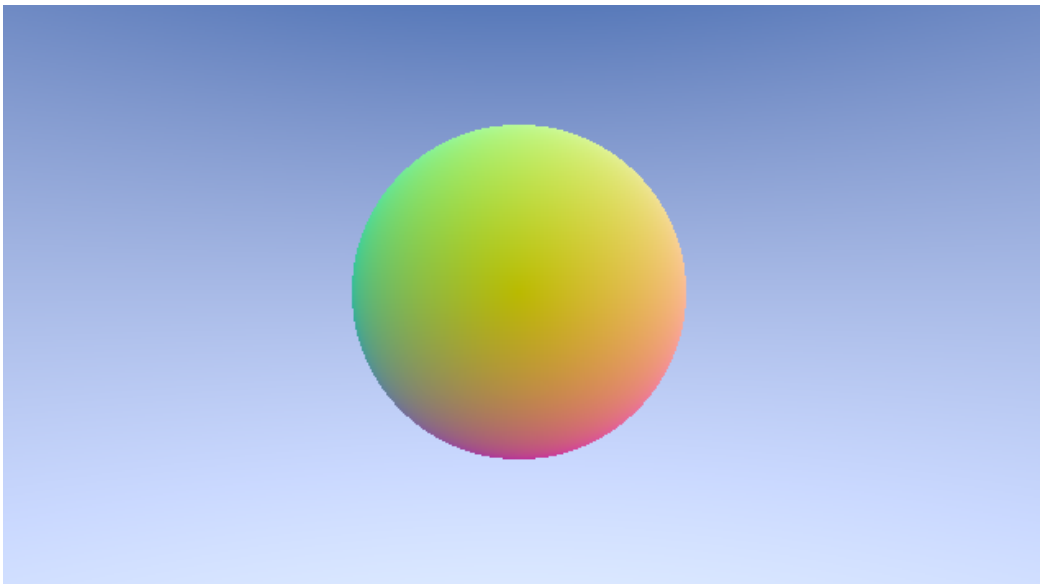


Σχήμα 3.8: Κάθετο διάνυσμα κορυφών για σφαίρα από τον Andreas Rejbrand *(c) copyright 2012-2017*

Το normal μιας επιφάνειας είναι ένα κάθετο διάνυσμα αυτής. Το normal χρειάζεται στους φωτισμούς για να γνωρίζει το πρόγραμμα πόσο φωτίζεται μια επιφάνεια. Χρησιμοποιείται επίσης στις αντανακλάσεις και στις διαθλάσεις για να ξέρει η ακτίνα τι κλίση θα πάρει.

```
vec3 GetNormal(vec3 pos)
{
    float c = sdf(pos).hitpoint;

    vec2 eps_zero = vec2(0.001, 0.0);
    return normalize(vec3(sdf(pos + eps_zero.xyy).hitpoint,
                          sdf(pos + eps_zero.yxy).hitpoint,
                          sdf(pos + eps_zero.yyx).hitpoint) - c);
}
```

Συνοπτικά ο τρόπος που λειτουργεί είναι ο εξής: Παίρνει δύο σημεία πάνω στην επιφάνεια, το ένα αυτό που χτύπησε η ακτίνα (pos) και το άλλο να είναι πολύ κοντά στο pos, pos+0.001 στην περίπτωση αυτή. Το κάνει και για τους τρεις άξονες εφόσον είναι τριών διαστάσεων για αυτό και εναλλάσσει την τιμή του x. Η λογική πίσω από αυτό είναι ότι έχοντας δύο σημεία πάνω σε ένα αντικείμενο, γίνεται να βρεθεί η εφαπτομένη όπου και επιστρέφει το κάθετο διάνυσμα της.



Σχήμα 3.9: Κανονικοποιημένα χρώματα με την χρήση των normals *(c) copyright 2018 Electric Square Ltd*

Τα χρώματα είναι κανονικοποιημένα από το 0 έως 1 και το normal παίρνει τιμές από το -1 έως 1. Δίνοντας για χρώμα το κανονικοποιημένο normal της επιφάνειας `color = normal * vec3(0.5)+ vec3(0.5)` η σφαίρα φαίνεται τρισδιάστατη. Άρα όταν το normal κοιτάζει πολύ στον άξονα y, τότε και το χρώμα του pixel παίρνει πολύ πράσινο. Για τους άξονες είναι (x,y,z) και για τα χρώματα (red,green,blue).

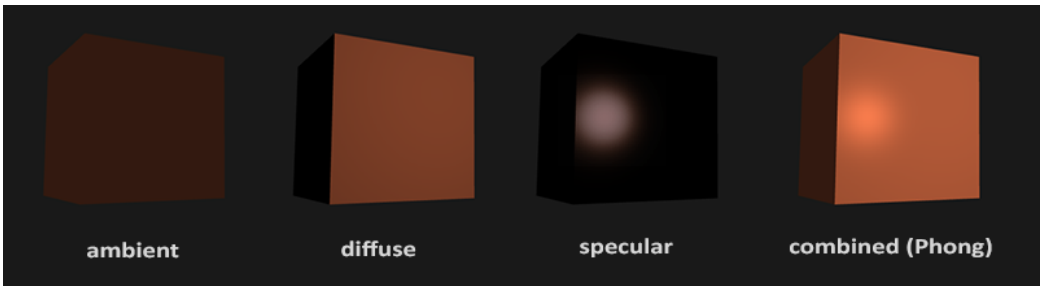
3.6 Σημειακή Πηγή Φωτός

Η σημειακή πηγή φωτός διαχέει την ίδια ποσότητα φωτός προς όλες τις κατευθύνσεις. Ένα παράδειγμα σημειακού φωτισμού είναι οι λάμπες. Το φως στον πραγματικό κόσμο βασίζεται σε πολλούς παράγοντες, κάτι το οποίο για να προσομοιωθεί θα ήταν πολύ ακριβό. Γι'αυτό τον λόγο η τεχνική αυτή χρησιμοποιεί 3 διαφορετικά συστατικά: **ambient**, **diffuse** και **specular**.

Το **ambient** είναι μία σταθερά η οποία δίνει πολύ λίγο φως σε όλα τα αντικείμενο στο χώρο. Ο λόγος που γίνεται αυτό είναι επειδή στον πραγματικό κόσμο είναι δύσκολο να έχεις απόλυτο σκοτάδι, διότι κατά πάσα πιθανότητα κάθε επιφάνεια θα δέχεται φως από κάποια μακρινή πηγή φωτός όπως το φεγγάρι, μία λάμπα από έναν μακρινό δρόμο κ.α.

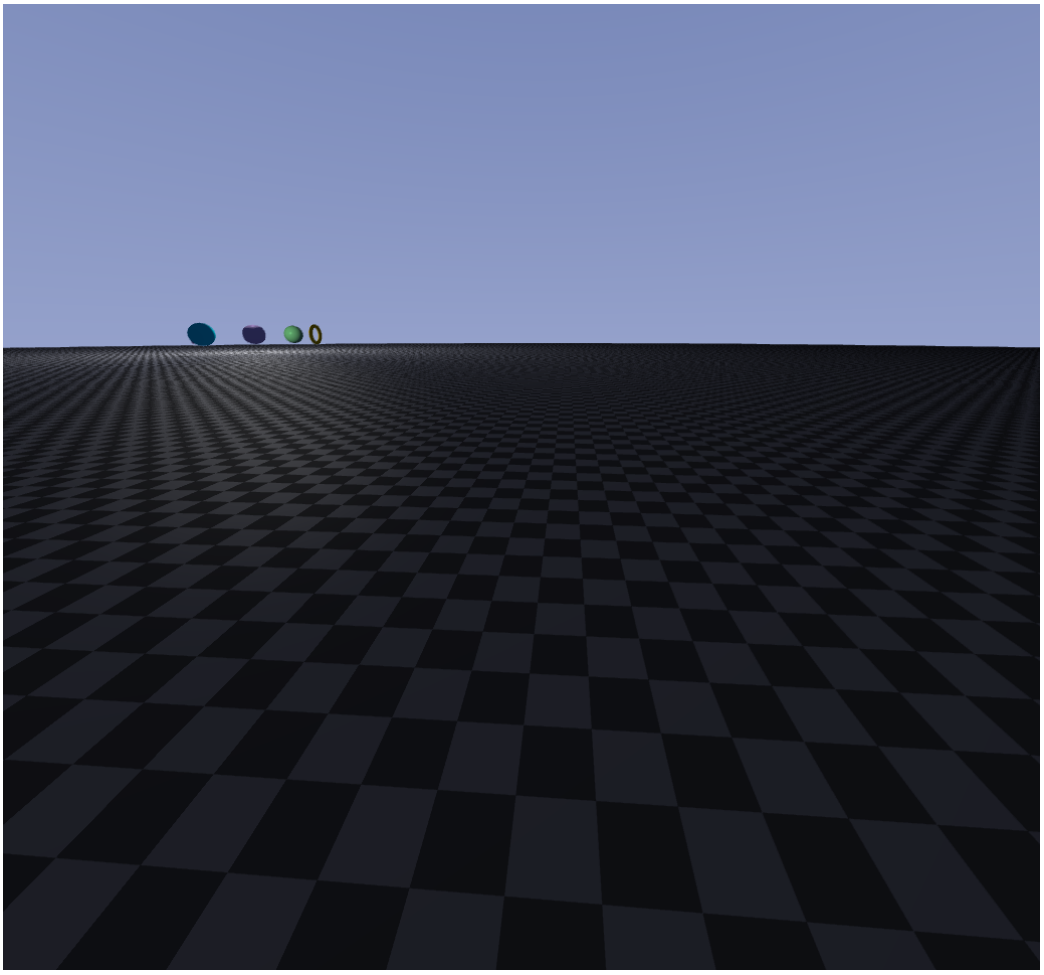
Το **diffuse** είναι το βασικότερο από όλα, αυτό που κάνει είναι να φωτίζει το αντικείμενο ανάλογα την κλίση που έχει με την ακτίνα φωτός. Χρησιμοποιεί τη γωνία που παράγεται από το normal της επιφάνειας και το διάνυσμα της ακτίνας του φωτός.

Το **specular** προσομοιώνει το φωτεινό σημείο που φαίνεται στα γυαλιστερά αντικείμενα. Ο specular φωτισμός δίνει περισσότερο από το χρώμα φωτός παρά του αντικειμένου γι'αυτό και το σημείο εκείνο τείνει να είναι πιο λευκό.



Σχήμα 3.10: Τρεία συστατικά για να παραχθεί ο Phong φωτισμός από τον By Joey de Vries *(c) copyright June 2014*

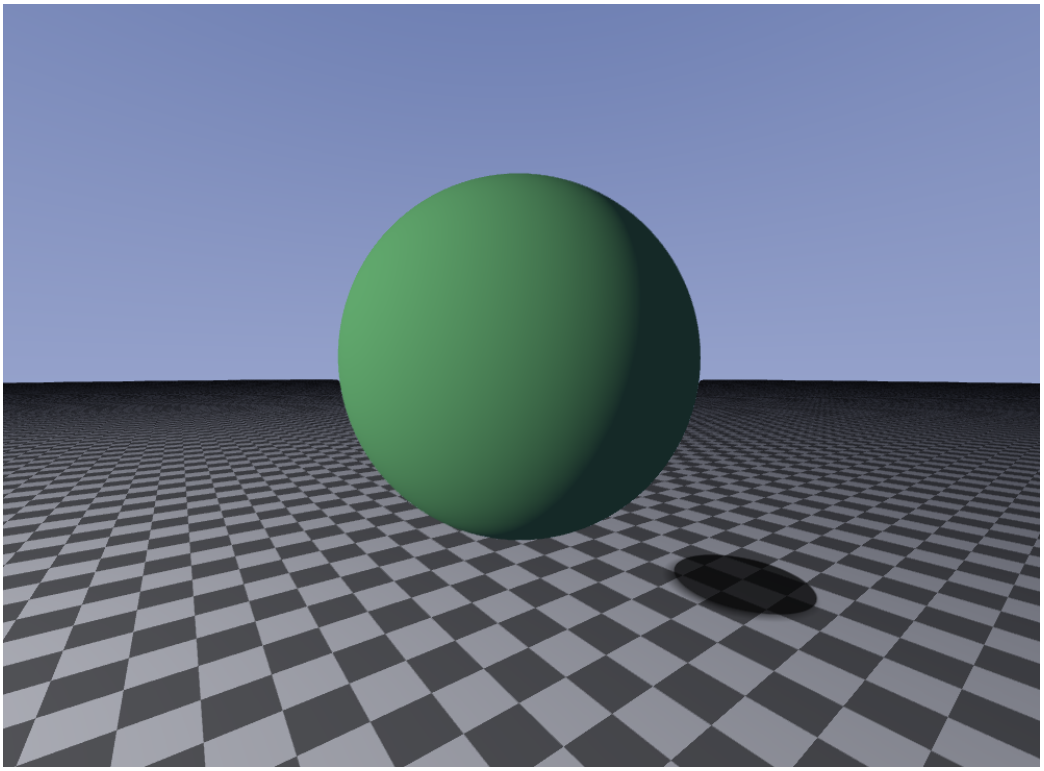
3.6.1 Περιβαλλοντικός Φωτισμός



Σχήμα 3.11: Περιβαλλοντικός φωτισμός στο πάτωμα.

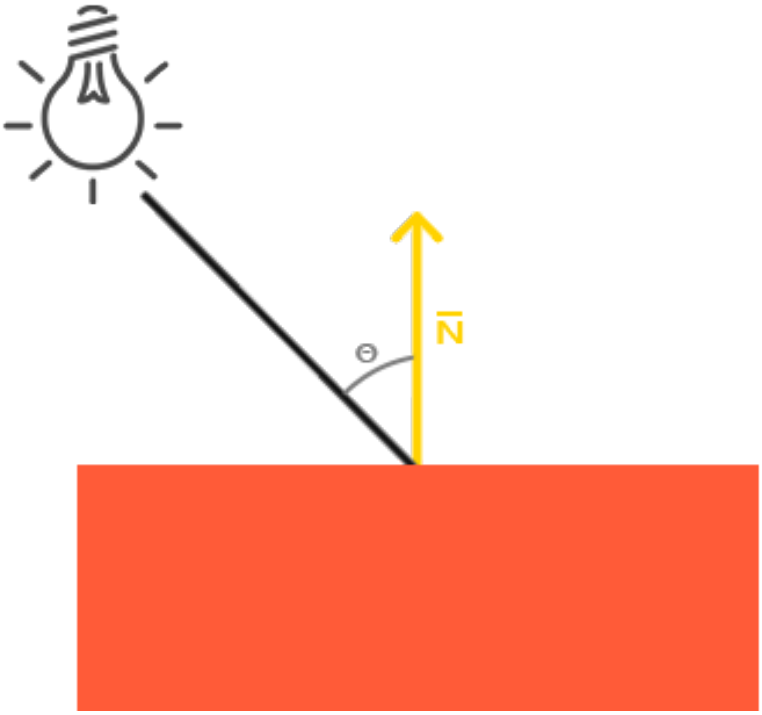
Το πάτωμα στην εργασία είναι ένα καλό παράδειγμα για την επίδειξη του περιβαλλοντικού φωτισμού. Από ένα σημείο και μετά το πάτωμα χωρίς ambient θα ήταν μαύρο, λόγο της εξασθένησης του φωτός (βλ. κεφάλαιο 3.4.4).

3.6.2 Διάχυτος Φωτισμός



Σχήμα 3.12: Διάχυτος φωτισμός και σκιάσεις.

Όπως φαίνεται και από την σκιά στο πάτωμα, το φως βρίσκεται λίγο πιο μπροστά και πάνω από την σφαίρα. Το diffuse είναι αυτό που δίνει φωτεινότητα στα αντικείμενα, όσο μικρότερη είναι η κλίση της γωνίας από την ακτίνα του φωτός και το normal της επιφάνειας τόσο πιο φωτεινό θα είναι και το σημείο εκείνο, δηλαδή όσο πιο κάθετα είναι το φως στην επιφάνεια. Η σκίαση στην σφαίρα είναι αποτέλεσμα του diffuse, γι'αυτό σιγά σιγά η σφαίρα γίνεται σκοτεινή. Να σημειωθεί ότι η σκιά στο πάτωμα είναι αποτέλεσμα διαφορετικής συνάρτησης και όχι του diffuse.



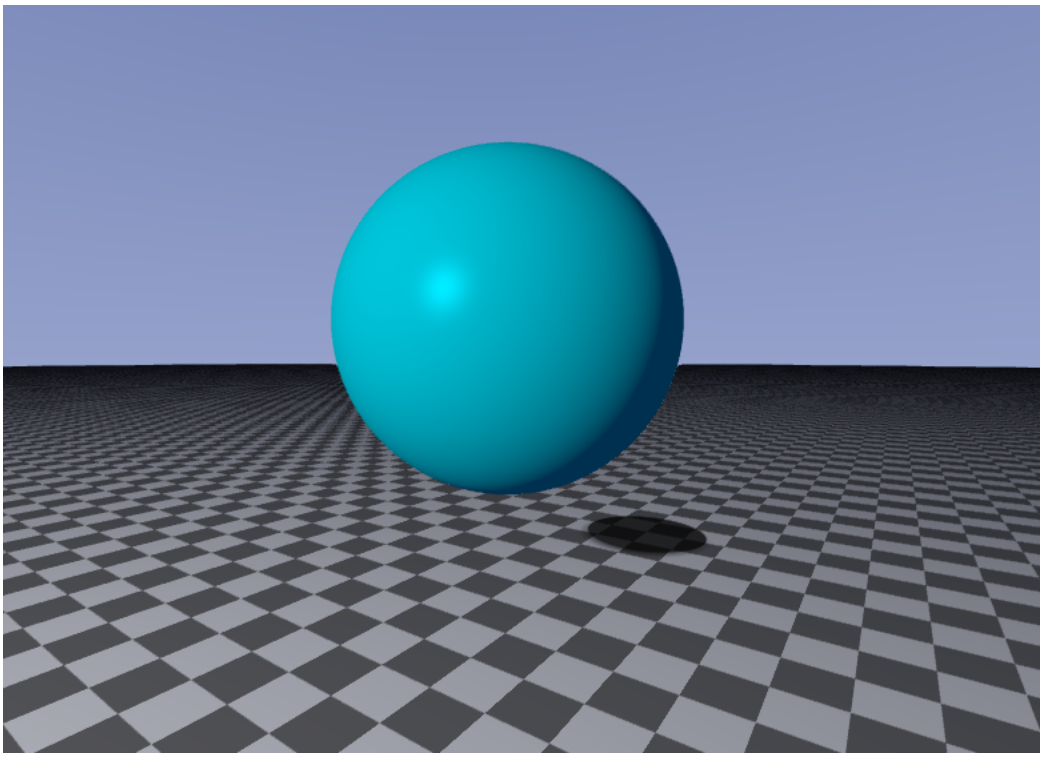
Σχήμα 3.13: Υπολογίζοντας τον διάχυτο φωτισμό από τον Joey de Vries *(c) copyright June 2014*

```
vec3 lightDir     = normalize(light.position - pos);
float NtoL_vector = max(dot(normal, lightDir), 0.0);
vec3 diffuse      = light.diffuse * NtoL_vector;
```

Παραπάνω αναπαριστάνεται ο τρόπος που υπολογίζεται ο φωτισμός. Να σημειωθεί ότι τα διανύσματα και τα χρώματα πρέπει να είναι κανονικοποιημένα αλλιώς μπορεί να εμφανιστούν περίεργα αποτελέσματα όπως "καμένα χρώματα".

Το lightDir είναι το διάνυσμα από το φως έως στο σημείο που χτύπησε η ακτίνα. Έπειτα πρέπει να υπολογιστεί η γωνία Θ, με το dot product υπολογίζεται η γωνία ανάμεσα σε δύο διανύσματα. Σε περίπτωση που η γωνία αυτή είναι μεγαλύτερη από 90 μοίρες το αποτέλεσμα του dot product θα γίνει αρνητικό με αποτέλεσμα να φωτίζονται σημεία που δεν θα έπρεπε. Η max το περιορίζει αυτό επιστρέφοντας 0 ως την μικρότερη πιθανή τιμή. Τέλος πολλαπλασιάζεται αυτό το ποσοστό με την φωτεινότητα που την ορίζει ο χρήστης.

3.6.3 Κατοπτρικός Φωτισμός



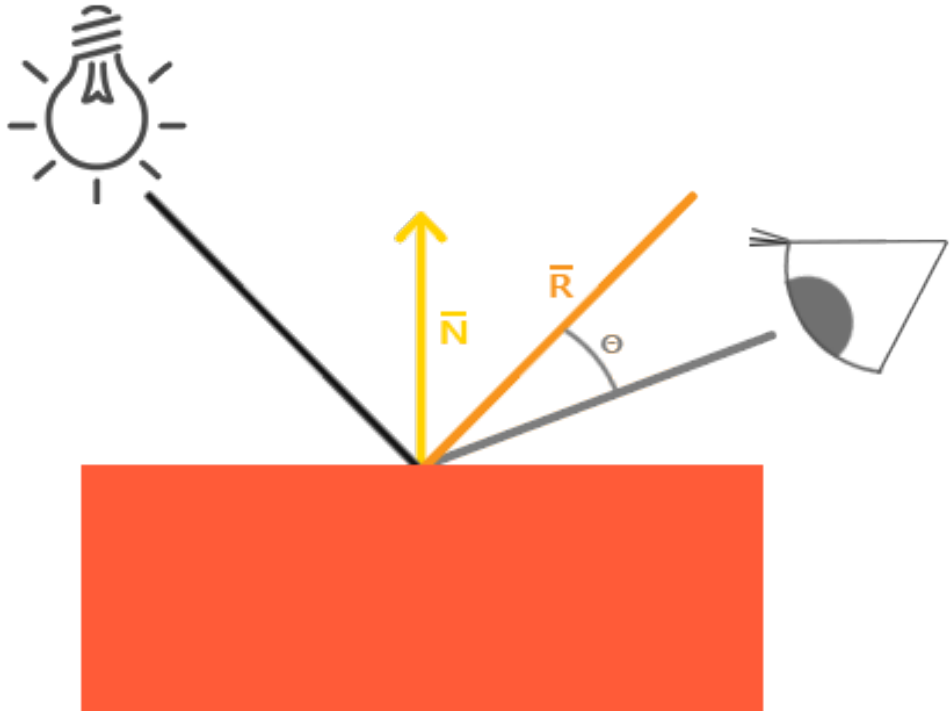
Σχήμα 3.14: Κατοπτρικός Φωτισμός

Το specular δίνει μία πιο γυαλιστερή υφή στο αντικείμενο σε σύγκριση με την εικόνα 3.9 που έχει μόνο diffuse και ambient. Ο φωτισμός παίζει μεγάλο ρόλο στην αναπαράσταση ενός υλικού. Στην παραπάνω εικόνα το specular lighting είναι αυτό που δημιουργεί το φωτεινό σημείο.

```
vec3 reflectDir   = reflect(lightDir, normal);
float spec        = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
vec3 specular     = light.specular * spec;
```

Το specular διαφέρει αρκετά στην υλοποίηση από το diffuse εφόσον χρειάζεται να γίνει αντανάκλαση της ακτίνας του φωτός προς την θέση της κάμερας. Η GLSL έχει δική της συνάρτηση reflect της οποίας ο κώδικας είναι ο παρακάτω, όπου I είναι το διάνυσμα της ακτίνας και N το normal της επιφάνειας.

$$I - 2.0 \times dot(N, I) \times N$$



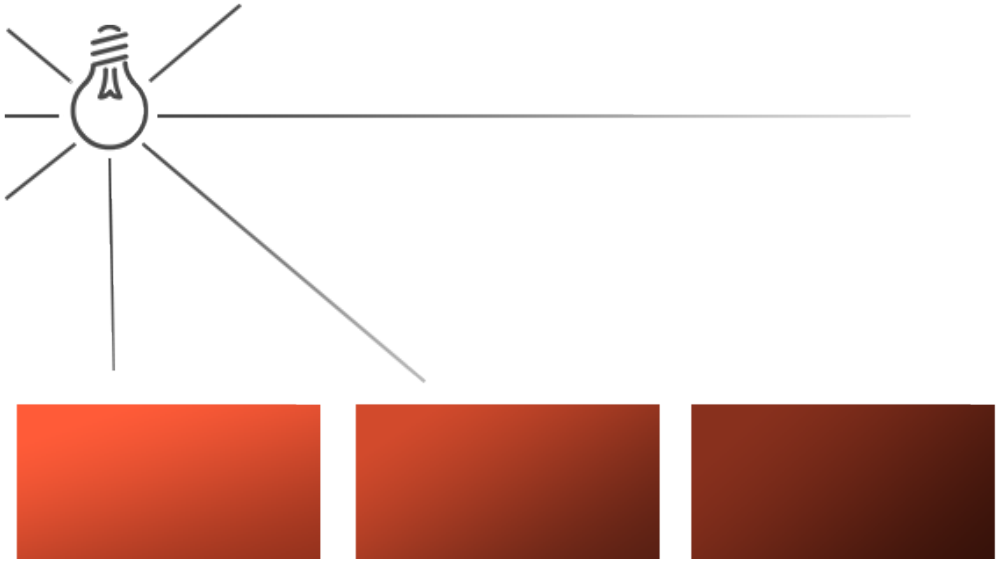
Σχήμα 3.15: Υπολογίζοντας τον κατοπτρικό φωτισμό από τον Joey de Vries *(c) copyright June 2014*

Όπως φαίνεται και στην εικόνα, στο specular θα χρειαστεί και η θέση της κάμερας. Όπως και στον πραγματικό κόσμο, για παράδειγμα σε ένα μαρμάρινο πάτωμα, όταν κινούμαστε βλέπουμε διαφορετικές ακτίνες φωτός που ανταναακλούν. Το ίδιο ισχύει και όταν σε μια πολύ λεία λίμνη ανταναακλάται το βουνό από πίσω, συμβαίνει το ίδιο πράγμα μόνο που η ακτίνα φωτός έχει πάρει χρώμα από το αντικείμενο που χτύπησε πριν.

```
vec3 viewDir      = normalize(pos - camera.pos.xyz);
vec3 reflectDir   = reflect(lightDir, normal);
float spec        = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
vec3 specular     = light.specular * spec;
```

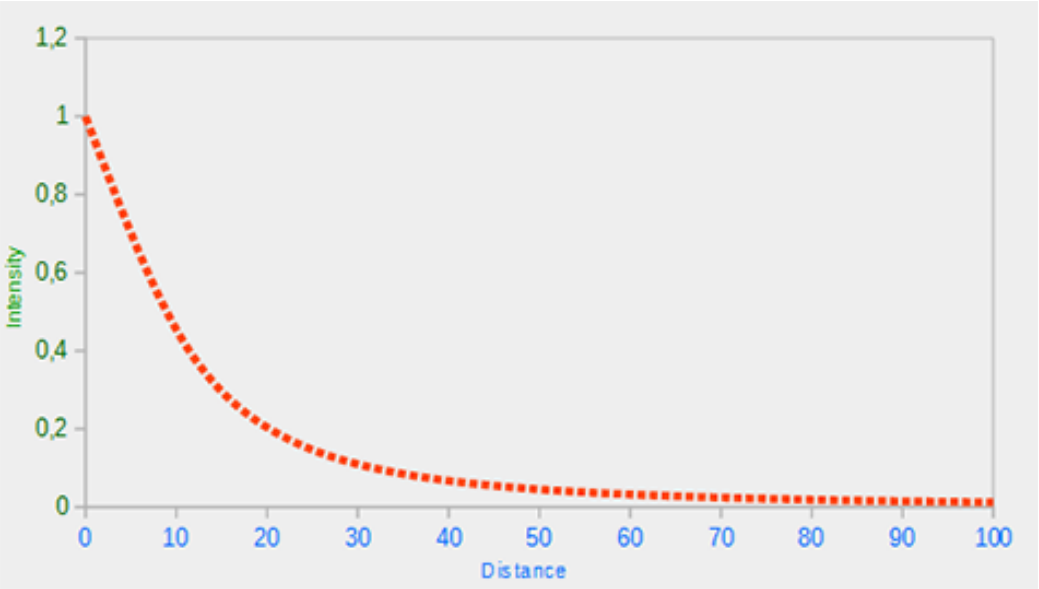
Η reflectDir είναι το διάνυσμα R στην εικόνα 3.12, έπειτα πρέπει να βρεθεί η γωνία Θ η οποία θα βρεθεί με το dot product του διανύσματος R και του διανύσματος από την κάμερα έως το πάτωμα. Όσο πιο κοντά στο R είναι το viewDir τόσο πιο φωτεινό θα είναι το αποτέλεσμα. Τέλος υψώνοντας το σε μία δύναμη αλλάζει η αντανάκλαση. Όσο μεγαλύτερος ο αριθμός που υψώνεται τόσο πιο γυαλιστερό θα είναι και το αντικείμενο.

3.6.4 Εξασθένηση Φωτός



Σχήμα 3.16: Υπολογίζοντας την εξασθένηση φωτός από τον Joey de Vries *(c) copyright June 2014*

Η δύναμη του φωτός πρέπει να μειώνεται καθώς η ακτίνα ταξιδεύει μακριά από την πηγή. Θα μπορούσε να γίνει με γραμμικό τρόπο, αλλά δεν θα φαίνεται ρεαλιστικό. Στον πραγματικό κόσμο για παράδειγμα μια λάμπα στο πεζοδρόμιο, το περισσότερο φως είναι ακριβώς κάτω από την λάμπα, δηλαδή ακόμα και μερικά μέτρα πιο πέρα έχει σχεδόν εξασθενήσει τελείως. Η παρακάτω γραφική παράσταση είναι αυτή που υλοποιείται στην εργασία.



Σχήμα 3.17: Γράφος εξασθένησης από τον Joey de Vries *(c) copyright June 2014*

Ο τύπος που προσομοιώνει την εξασθένηση για το φως της σημειακής πηγής είναι ο εξής:

$$F_{att} = \frac{1.0}{K_c + K_l \times d + K_q \times d^2} \tag{3.1}$$

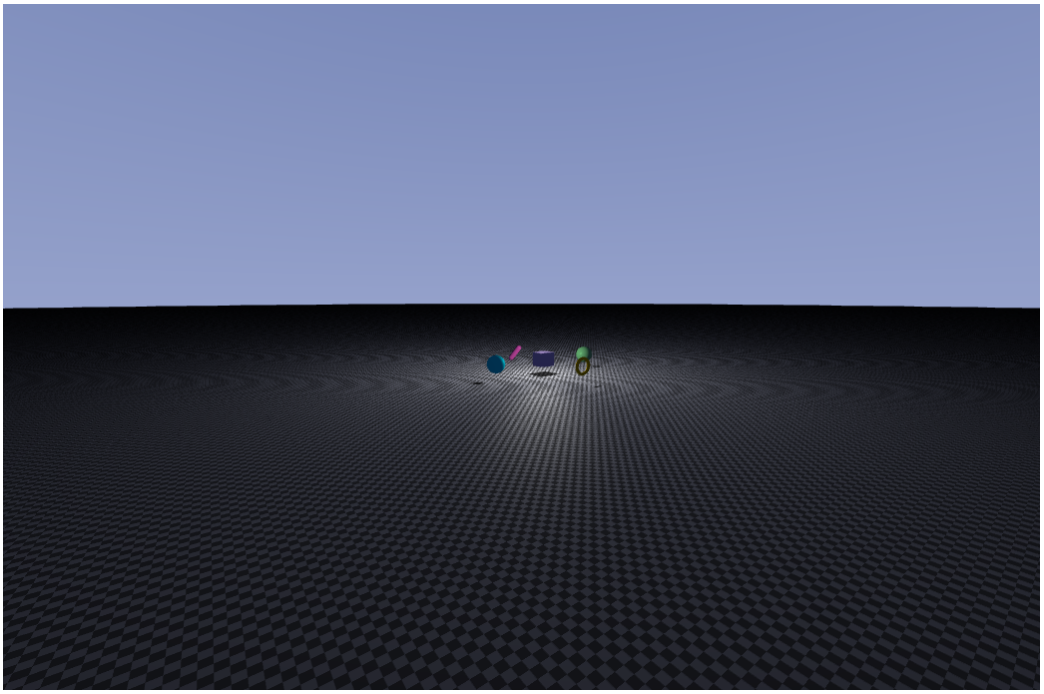
Το **d** είναι η απόσταση που διανύει η ακτίνα, η **Kc** είναι μία σταθερά, η **Kl** δίνει την γραμμικότητα στην εξίσωση και η **Kq** δίνει την απότομη ρήξη της δύναμης του φωτός.

Η σταθερά **Kc** είναι 1 γιατί σε περιπτώσεις που η απόσταση γίνει πολύ μικρή, ο παρονομαστής θα γίνει και αυτός πολύ μικρός με αποτέλεσμα να "καίει" τα χρώματα.
Η **Kl** πολλαπλασιάζεται με την απόσταση και είναι αυτή που δίνει την γραμμικότητα στην παράσταση.
Η **Kq** πολλαπλασιάζεται με την απόσταση στο τετράγωνο, αυτό έχει σαν αποτέλεσμα η **Kq** να μικραίνει απότομα στην αρχή και όσο μεγαλώνει η απόσταση μικραίνει, με αποτέλεσμα στην αρχή το φως να χάνει γρήγορα την δύναμη του.

Όλοι οι υπολογισμοί για το σημειακό φως γίνονται μέσα στην ίδια συναρτήση.

```
float distance    = length(light.position - pos);
float attenuation = 1.0 / (light.constant + light.linear * distance + light.
    quadratic * (distance * distance));
diffuse  *= attenuation;
ambient  *= attenuation;
specular *= attenuation;

vec3 final = color * (diffuse + ambient + specular);
return final;
```

Αφού υπολογιστεί το ποσοστό της εξασθένησης πολλαπλασιάζεται και με τα 3 στοιχεία του σημειακού φωτός και τέλος πολλαπλασιάζει την δυναμικότητα του φωτός με το χρώμα της επιφάνειας προτού το επιστρέψει. Από μακριά το φως φαίνεται όντως σαν να είναι από μία κοντινή πηγή φωτός. Επίσης χάρη στο ambient το πάτωμα το οποίο είναι ουσιαστικά άπειρο δεν θα γίνει ποτέ τελείως μαύρο.



Σχήμα 3.18: Σημειακή πηγή φωτός με εξασθένηση φωτός

3.7 Κινούμενη κάμερα

Για την κίνηση της κάμερας, το πρόγραμμα πρέπει να δέχεται πληροφορίες από το πληκτρολόγιο και το ποντίκι. Η βιβλιοθήκη GLFW που χρησιμοποιείται στην διατριβή αυτή, παρέχει συναρτήσεις που επιτρέπουν την επικοινωνία αυτή.

3.7.1 Θέση Κέρσορα

Με την συνάρτηση `glfwSetCursorPosCallback(window, mouse_callback)` ορίζεται η συνάρτηση η οποία θα καλείται σε κάθε καρέ και θα διαβάζει πληροφορίες από το ποντίκι. Παρακάτω η συνάρτηση που επιστρέφει την θέση του κέρσορα:

```
void mouse_callback(GLFWwindow *window, double xpos, double ypos)
{
    if (firstMouse) {
        lastX      = xpos;
        lastY      = ypos;
        firstMouse = false;
    }

    float xoffset = lastX - xpos;
    float yoffset = lastY - ypos;
    lastX         = xpos;
    lastY         = ypos;

    mouse.ProcessMouseOffset(xoffset, yoffset);
    camera.setMouse(-xpos, -ypos);
}
```

Ο πρώτος έλεγχος γίνεται μόνο για τον πρώτο καρέ, ώστε όταν ξεκινάει το πρόγραμμα να μην δημιουργεί ένα περίεργο τρέμουλο, γι'αυτό και στην προηγούμενη θέση του κέρσορα μπαίνει η παρόν. Έπειτα υπολογίζεται η απόσταση που δημιουργήθηκε από το προηγούμενο καρέ. Το offset δεν χρειάζεται σε αυτήν την κάμερα και υπάρχει μόνο για πειραματισμούς χρωμάτων στα αντικείμενα. Τέλος στέλνει την νέα θέση του κέρσορα στην κλάση της κάμερας.

3.7.2 Εισαγωγή Τιμών Πληκτρολογίου

Η εισαγωγή από το πληκτρολόγιο είναι λίγο διαφορετική, εφόσον δεν χρειάζεται να οριστεί ειδική συνάρτηση στην GLFW που θα καλείται για κάθε καρέ, ωστόσο δημιουργώντας μία συνάρτηση και βάζοντας την μέσα στην game loop γίνεται ακριβώς το ίδιο πράγμα.

Η συνάρτηση proccessInput καλείται στην αρχή του καρέ προτού κληθεί κάποιο άλλο συμβάν:

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        zaxisNeg = true;
    else {
        zaxisNeg = false;
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        zaxisPos = true;
    else {
        zaxisPos = false;
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        xaxisPos = true;
    else {
        xaxisPos = false;
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        xaxisNeg = true;
    else {
        xaxisNeg = false;
    }
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS && glfwGetKey(window,
        GLFW_KEY_A) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_D) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_S) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_D) == GLFW_PRESS) {
        halfSpeed = true;
    }
    else {
        halfSpeed = false;
    }
    camera.lookAt(zaxisNeg, zaxisPos, xaxisNeg, xaxisPos, halfSpeed,
        deltaTime);
}
```

Από ότι φαίνεται παραπάνω οι μεταβλητές είναι όλες τύπου boolean, διότι στο τέλος το μόνο που θα αλλάζει θα είναι το διάνυσμα στο οποίο θα κινείται η κάμερα ανάλογα τα πλήκτρα που πατηθούν. Σε περίπτωση που πατηθούν δύο μαζί τότε η ταχύτητα γίνεται μισή, γιατί προστίθεται. Τέλος στέλνονται όλες οι μεταβλητές στην συνάρτηση lookAt της κλάσης camera.

Στις περιπτώσεις που κάποιο πλήκτρο πρέπει να κληθεί μόνο την φορά που πατηθεί και όχι για κάθε καρέ, υπάρχει ειδική συνάρτηση από την GLFW που το χειρίζεται αυτό, με την `glfwSetKeyCallback(window, key_callback);`.

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int
     mods)
{
    if (key == GLFW_KEY_UP && action == GLFW_PRESS)
        if (bounce < 5)
            bounce += 1;
    if (key == GLFW_KEY_DOWN && action == GLFW_PRESS)
        if (bounce > 0)
            bounce -= 1;

    if (key == GLFW_KEY_F1 && action == GLFW_PRESS)
        AA = !AA;

    if (key == GLFW_KEY_L && action == GLFW_PRESS)
    {
        if (showQuad)
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        else
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        showQuad = !showQuad;
    }
}
```

Η συνάρτηση αυτή χρησιμοποιείται κυρίως για τον αριθμό αντανακλάσεων της ακτίνας και για να ανοίγει και να κλείνει το anti-aliasing. Χωρίς την χρήση μίας τέτοιας συνάρτησης όταν θα πατιόταν για παράδειγμα το πλήκτρο που ανεβάζει τις ανταναακλάσεις, η πρόσθεση θα γινόταν για κάθε καρέ, δηλαδή μέχρι και 60 φορές το δευτερόλεπτο. Το ίδιο και με το anti-aliasing, θα ήταν σαν γινόταν coin flip για το άμα θα μείνει ανοιχτό ή κλειστό.

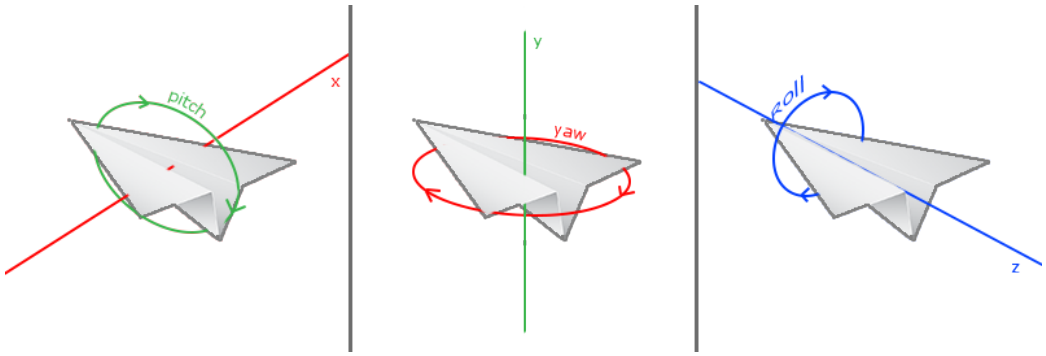
3.7.3 Κλάση της Κάμερας

Η κλάση της κάμερας χρησιμοποιείται για να δημιουργεί το αντικείμενο, επίσης έχει κατασκευαστή που δέχεται ορίσματα όπως την ταχύτητα της κάμερας και την ευαισθησία του κέρσορα. Έχει δύο συναρτήσεις, μία δέχεται την θέση του κέρσορα και η άλλη δημιουργεί τους νέους άξονές της.

Πρώτα πολλαπλασιάζει την θέση του κέρσορα στον κάθε άξονα με την ευαισθησία του:

```
angleX = xpos * mouseSensitivity;
angleY = ypos * mouseSensitivity;
```

Από την στιγμή που η κάμερα θα περιστρέφεται μόνο σε pitch και yaw, δεν χρειάζεται να υπολογιστεί ο άξονας z.



Σχήμα 3.19: Camera Pitch Yaw Roll By Joey de Vries *(c) copyright June 2014*

Έπειτα φτιάχνει τον πίνακα περιστροφής για τον κάθε άξονα, το πρώτο όρισμα είναι ένας άδειος πίνακας, το δεύτερο η γωνία μετατρεπόμενη σε ακτίνια και το τρίτο είναι ο άξονας περιστροφής. Pitch και yaw αντίστοιχα.

```
mat4 rotateX = glm::rotate(mat4(1.0), glm::radians(angleY), vec3(1.0, 0.0,
    0.0));
mat4 rotateY = glm::rotate(mat4(1.0), glm::radians(angleX), vec3(0.0, 1.0,
    0.0));
mat4 rotationMatrix = rotateY * rotateX;
```

Εφόσον έχει υπολογιστεί η κλίση της κάμερας με το ποντίκι, επόμενο βήμα είναι να υπολογιστούν ξανά οι άξονες της κάμερας.

```
forward = glm::normalize(vec3(rotationMatrix * vec4(0.0, 0.0, -1.0, 0.0)));
up      = glm::normalize(vec3(rotationMatrix * vec4(0.0, 1.0, 0.0, 0.0)));
right   = glm::normalize(glm::cross(forward, up));
```

Πολλαπλασιάζει το matrix με τύπο vec4 ο οποίος κοιτάζει μπροστά και μετά όλο αυτό μέσα σε vec3 ώστε να επιστραφούν μόνο οι x,y,z τιμές. Δεν γίνεται μετατροπή ενός 4x4 matrix σε vec3 απευθείας. Η ίδια λογική ισχύει και για τον y άξονα. Τέλος βρίσκει και το διάνυσμα του x άξονα με το cross product.

Μέχρι τώρα βρέθηκε το σημείο που θα κοιτάζει η κάμερα και έχει μείνει να βρεθεί η νέα θέση της.

```
if (zN) // W
    cameraPos += (keyboardSpeed * forward) * deltaTime;
if (zP) // S
    cameraPos += (keyboardSpeed * (-forward)) * deltaTime;
if (xN) // A
    cameraPos += (keyboardSpeed * (-right)) * deltaTime;
if (xP) // D
    cameraPos += (keyboardSpeed * right) * deltaTime;
```

Στην θέση της κάμερας προστίθεται το διάνυσμα στο οποίο θα κινηθεί επί την ταχύτητα του πληκτρολογίου και όλο αυτό μαζί με την deltaTime η οποία θα εξηγηθεί στο κεφάλαιο 3.7.4.

Σε περίπτωση που πατηθούν δύο πλήκτρα ταυτόχρονα, το keyboardSpeed θα γίνει μισό και η κάμερα θα κινηθεί προς την κατεύθυνση των δύο πλήκτρων που πατήθηκαν.

```
if (halfSpeed) {
    keyboardSpeed = 5.0;
}
else
    keyboardSpeed = 10.0;
```

3.7.4 Delta Time

Ένα πρόβλημα που υπάρχει σε όλες τις μηχανές γραφικών είναι το εξής: Εφόσον η συνάρτηση για το πληκτρολόγιο καλείται μέσα στην game loop, αυτό σημαίνει ότι η θέση της κάμερας θα κινηθεί προς μια κατεύθυνση όσες φορές παράγει η κάρτα γραφικών κάποιο καρέ. Αυτό σημαίνει ότι η κάμερα στις κάρτες που παράγουν λιγότερα καρέ θα κινείται πιο αργά και όταν θα παράγονται πολλά καρέ το αντίθετο. Η Delta Time είναι μία μεταβλητή που γίνεται μικρότερη όσα περισσότερα καρέ παράγονται και μεγαλύτερη όσο λιγότερα. Παρακάτω η υλοποίηση της.

```
double lastTime      = 0.0;
// Game loop
while (!glfwWindowShouldClose(window)) {
    float currentFrame = glfwGetTime();
    deltaTime          = currentFrame - lastFrame;
    lastFrame          = currentFrame;
    ...
    ...
    ...
}
```

Η `glfwGetTime();` επιστρέφει τον χρόνο σε δευτερόλεπτα από την στιγμή που η glfw έχει γίνει initialized. Έπειτα αφαιρεί τον παρόν χρόνο με τον χρόνο από την τελευταία φόρα που παράχθηκε καρέ (την πρώτη φορά προφανώς θα είναι 0). Με αυτόν τον τρόπο όταν παράγει πολλά καρέ, η αφαίρεση αυτή επιστρέφει μικρό αριθμό και το αντίθετο στην άλλη περίπτωση. Έτσι πολλαπλασιάζοντας, την θέση της κάμερας με αυτόν τον αριθμό, είτε την καθυστερεί ή την επιταχύνει.

Έπειτα στέλνονται οι νέες τιμές της κάμερας στον shader με uniforms.

```
setVec4(marching, "camera.pos", camera.cameraPos.x, camera.cameraPos.y,
    camera.cameraPos.z, 0.0);
setVec4(marching, "camera.dir", camera.forward.x, camera.forward.y, camera.
    forward.z, 0.0);
setVec4(marching, "camera.yAxis", camera.up.x, camera.up.y, camera.up.z,
    0.0);
setVec4(marching, "camera.xAxis", camera.right.x, camera.right.y, camera.
    right.z, 0.0);
```

Οι συναρτήσεις setVec4 υπάρχουν για απλοποίηση, υπάρχει διαφορετική διαδικασία από πίσω.

Μέσα στον compute shader υπάρχει struct για την κάμερα και δέχεται έτσι της τιμές της κάμερας.

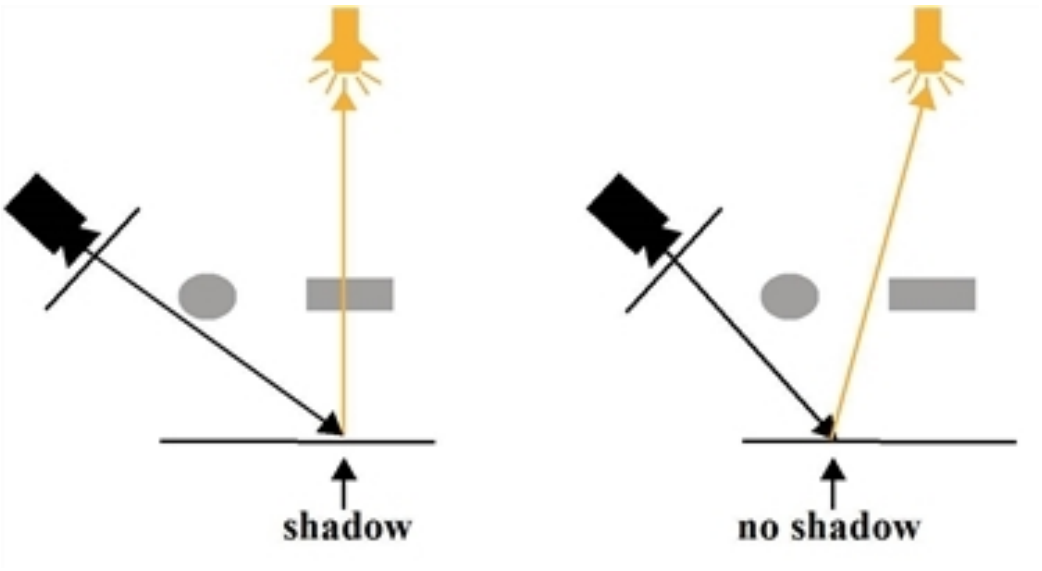
```
struct Camera {
    vec4 pos;
    vec4 dir;
    vec4 yAxis;
    vec4 xAxis;
};
uniform Camera camera;
```

Πλέον η συνάρτηση που πετάει την ακτίνα μέσα στον shader θα είναι διαφορετική, εφόσον το μόνο που έχει να κάνει είναι να πετάξει ακτίνα από το pixel που κάνει render, με την καινούρια θέση της κάμερας.

```
Ray castRay(vec2 uv)
{
    float Perpective = radians(45);
    vec4 dir         = normalize(uv.x * camera.xAxis + uv.y * camera.yAxis +
        camera.dir * Perpective);

    return Ray(camera.pos.xyz, dir.xyz);
}
```

3.8 Σκιές

Σε προηγούμενο κεφάλαιο παρουσιάστηκε ο τρόπος που ο φωτισμός δίνει σκίαση στα αντικείμενα. Ωστόσο η διαδικασία σκίασης από ένα αντικείμενο σε ένα άλλο γίνεται με διαφορετικό τρόπο. Ο τρόπος αυτός μοιάζει πολύ με την διαδικασία που ακολουθούν οι αρχικές ακτίνες.

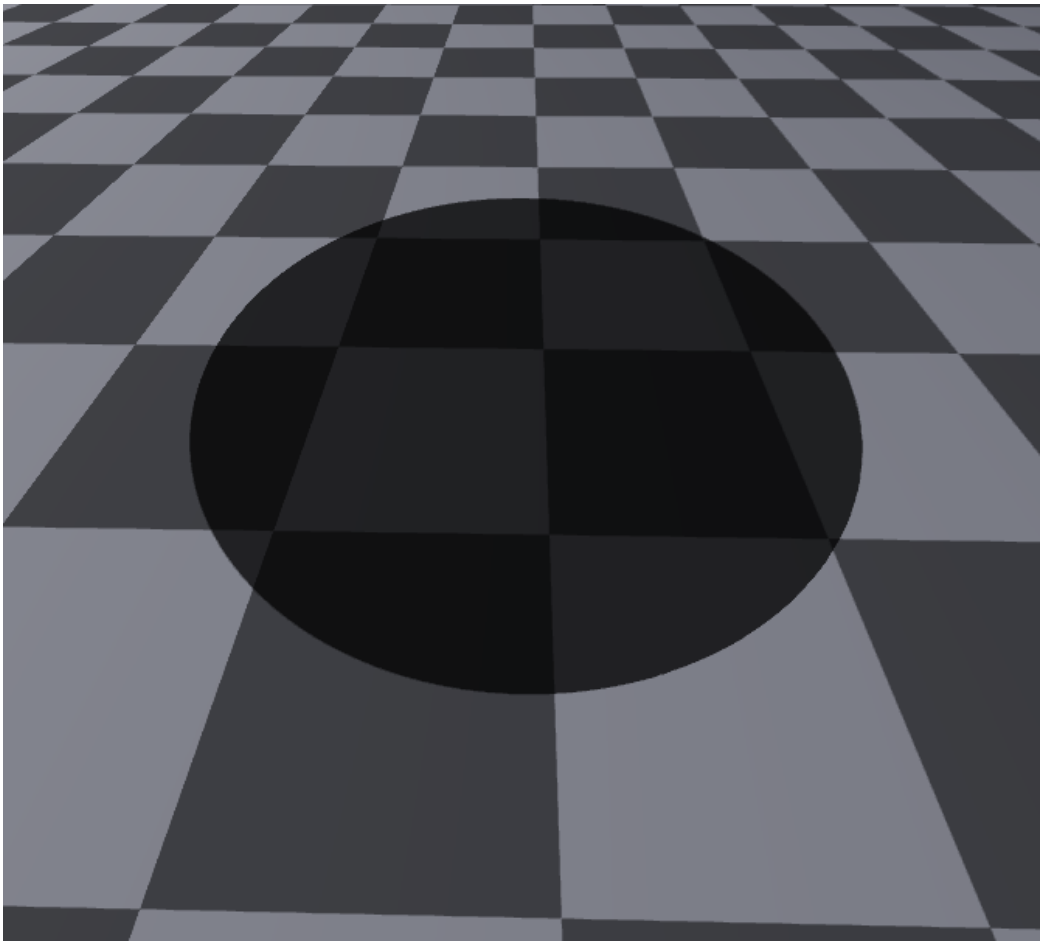


Σχήμα 3.20: Υπολογίζοντας σκιές με Ray Marching

3.8.1 Σκληρές Σκιές

Όταν μία αρχική ακτίνα χτυπήσει για παράδειγμα το πάτωμα, τότε εκείνο το σημείο πρέπει να εξετασθεί άμα είναι σε σκίαση. Ο τρόπος που λειτουργεί αυτό είναι κάνοντας ray marching από το σημείο που χτύπησε η αρχική ακτίνα μέχρι το φως. Άμα υπάρχει αντικείμενο ενδιάμεσα τότε το σημείο που χτύπησε η ακτίνα είναι σε σκιά.

```
float softshadow(vec3 ro, vec3 rd)
{
    for (float t = 0.0; t < 1;)
    {
        RayHit h = sdf(ro + rd * t);
        if(h.hitpoint < 0.001)
            return 0.05;

        t += h.hitpoint;
    }

    return 1.0;
}
```

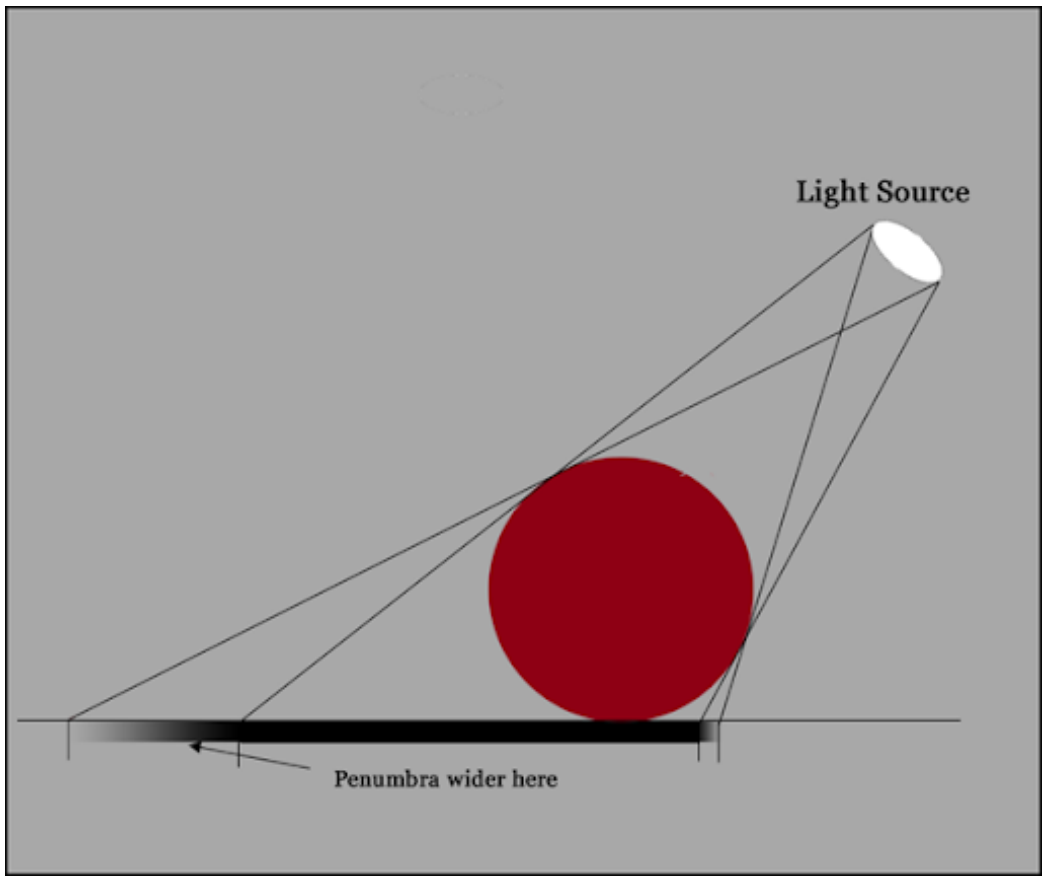


Σχήμα 3.21: Σκληρή σκιά από μία σφαίρα

Η συνάρτηση sdf ορίζει όλα τα αντικείμενα στον κόσμο, επιστρέφει κάθε φορά τα χαρακτηριστικά του κοντινότερου αντικειμένου, στην περίπτωση αυτή χρειάζεται μόνο την απόσταση του. Όταν η απόσταση με το αντικείμενο που εξέτασε είναι πολύ μικρή (0.001) σημαίνει ότι το σημείο που εξετάζει είναι υπό σκιά, άρα επιστρέφει μία πολύ μικρή τιμή για να το πολλαπλασιάσει με αυτήν. Η τιμή δεν είναι 0 για να κρατήσει λίγο από το χρώμα του.

3.8.2 Απαλές Σκιές

Σε αυτήν την εργασία χρησιμοποιούνται απαλές σκιές, οι οποίες προσομοιώνουν ένα φαινόμενο που συμβαίνει στον πραγματικό κόσμο και λέγεται **penumbra**.



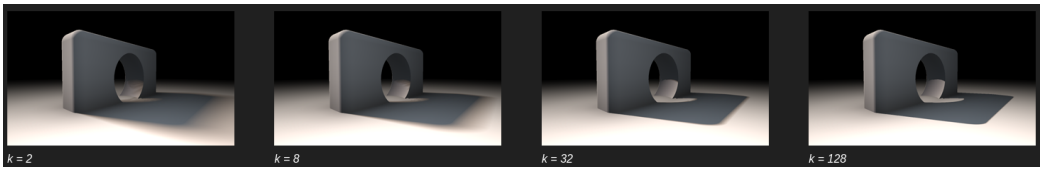
Σχήμα 3.22: Μήκος της penumbra από τον David Briggs και Ray Kristanto *(c) copyright 2007*

Το κόλπο για την προσομοίωση της penumbra είναι να αποφασιστεί τι θα γίνει με τις ακτίνες φωτός που είναι πολύ κοντά στο να χτυπήσουν το αντικείμενο. Όπως φαίνεται και στην εικόνα ο φωτισμός είναι σαν να κάνει γραμμική παρεμβολή ανάμεσα στα δύο σημεία που βρήκαν οι ακτίνες.

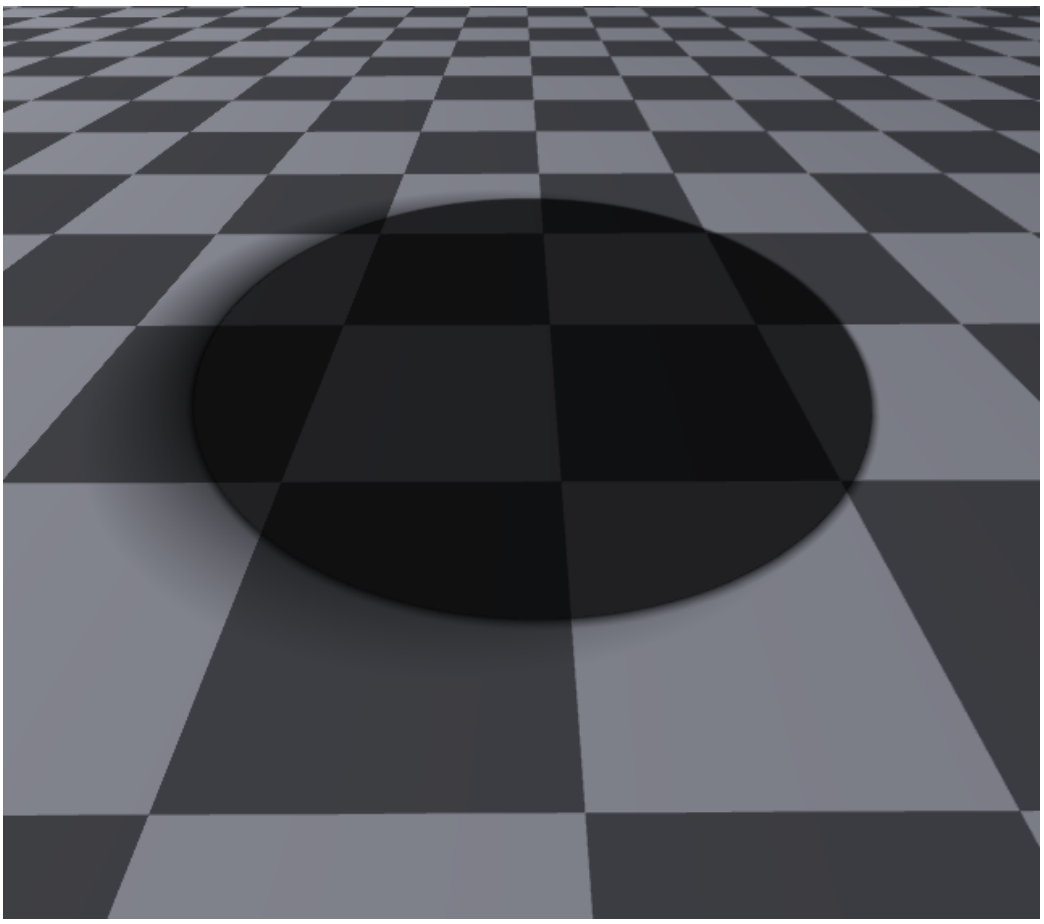
Η συνάρτηση είναι σχεδόν ίδια με την προηγούμενη μόνο με μία διαφορά.

```
float softshadow(vec3 ro, vec3 rd, float k)
{
    float res = 1.0;
    for (float t = 0.0; t < 1;)
    {
        RayHit h = sdf(ro + rd * t);
        if(h.hitpoint < 0.001)
            return 0.05;

        res = min(res, k * h.hitpoint / t);
        t += h.hitpoint;
    }

    return res;
}
```

Όσο πιο κοντά ήταν στο να πετύχει η ακτίνα το αντικείμενο τόσο πιο σκιερή πρέπει να είναι η σκιά και το ίδιο να συμβαίνει όταν το σημείο αυτό της penumbra είναι κόντα σε κάποιο που είναι σίγουρα σε σκιά. Αυτά τα δύο σημεία είναι το **h** και το **t**. Όσα μικρότερη είναι η απόσταση από την ακτίνα στο αντικείμενο, δηλαδή το **h**, τόσο μικρότερο και το αποτέλεσμα της διαίρεσης. Η μεταβλητή **k** ουσιαστικά ορίζει την σκληρότητα της σκιας. Στην εικόνα 3.20 δίνονται μερικά παραδείγματα με διάφορες τιμές για την **k**. Στην εργασία αυτή, η **k** έχει τιμή 2.



Σχήμα 3.23: Σκληρότητα σκιών από τον Inigo Quilez *(c) copyright 2018*



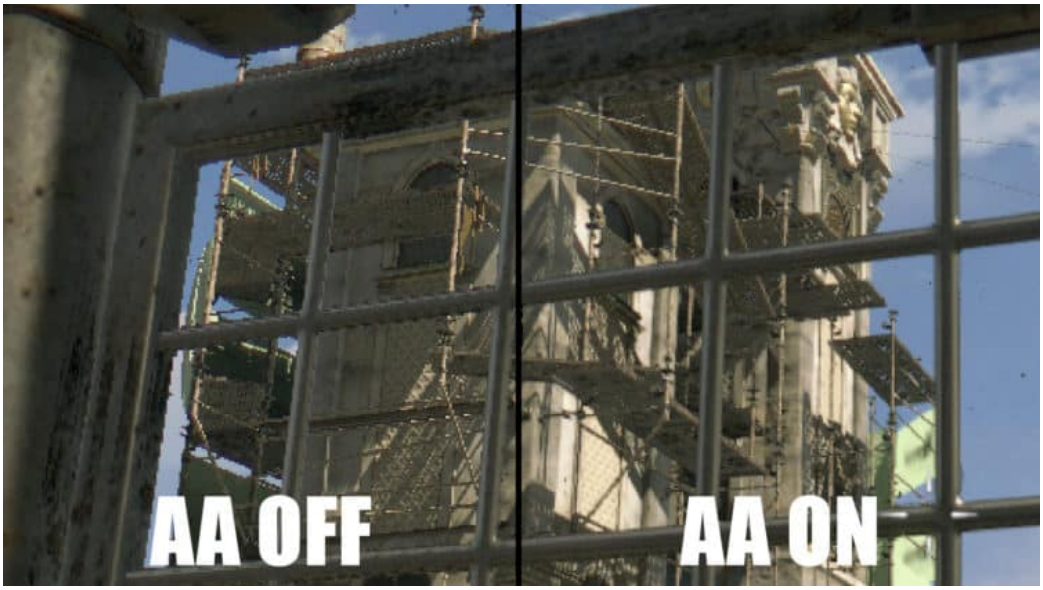
Σχήμα 3.24: Απαλή σκιά από μία σφαίρα

Παραπάνω η απεικόνιση απαλής σκιάς. Όπως και στο παράδειγμα στην εικόνα 3.19 το μήκος της penumbra είναι μεγαλύτερο όταν η ακτίνα φωτός που σχεδόν αγγίζει το αντικείμενο είναι μεγαλύτερη.

Σημείωση: *Τα βήματα που γίνονται για την εξέταση της σκιάς είναι πολύ λίγα και κατά πάσα πιθανότητα σε μια διαφορετική σκηνή με μεγαλύτερες αποστάσεις, χρειαζόταν να ανέβει ο αριθμός των μέγιστων βημάτων. Αυτό έγινε για λόγους απόδοσης.*

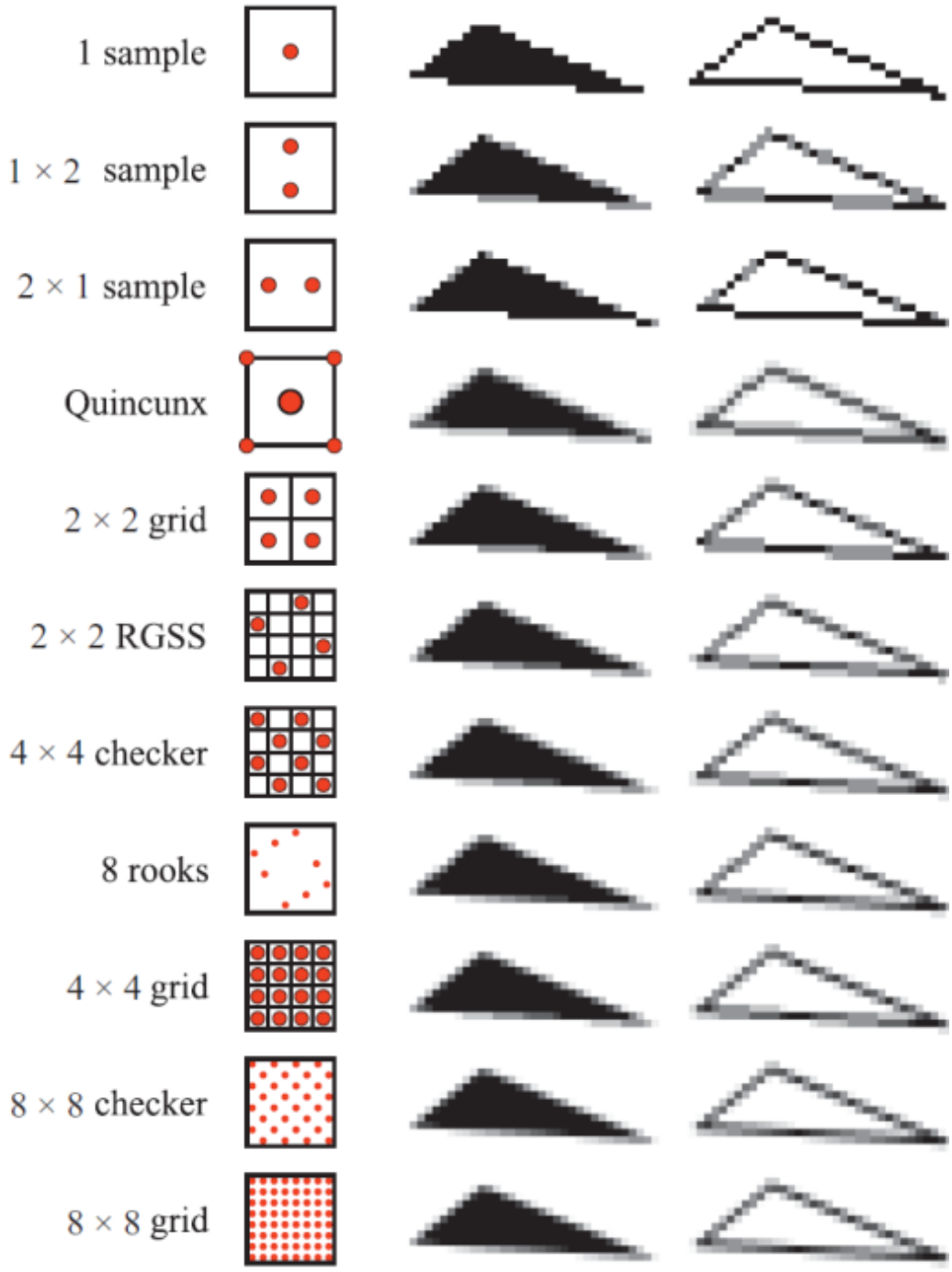
3.9 Anti Aliasing

Στα γραφικά υπολογιστών υπάρχει ένα πρόβλημα το οποίο λέγεται aliasing. Αυτό που συμβαίνει είναι να αποκόβει τα αντικείμενα απότομα κάνοντας εμφανή τα pixel της οθόνης.



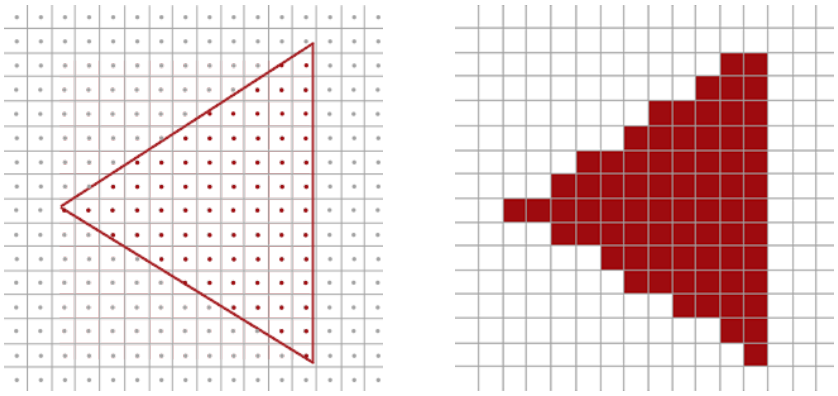
Σχήμα 3.25: Anti Aliasing On vs Off By Brent Hale *(c) copyright January 2019*

Στην παραπάνω εικόνα, όταν το anti aliasing είναι κλειστό γίνονται εμφανή τα pixels, σαν σκαλοπάτια. Σε περιπτώσεις που το αντικείμενο στο video game είναι κάθετο ως προς τα pixels της οθόνης δεν φαίνεται αυτό το άσχημο αποτέλεσμα. Υπάρχουν πολλοί μέθοδοι για το anti aliasing, ωστόσο δεν γνωρίζουμε ποια είναι η καλύτερη εφόσον η καθεμία έχει τα θετικά και αρνητικά της.



Σχήμα 3.26: Supersampling patterns By Real-Time Rendering, 3rd Edition, A K Peters *(c) copyright 2008*

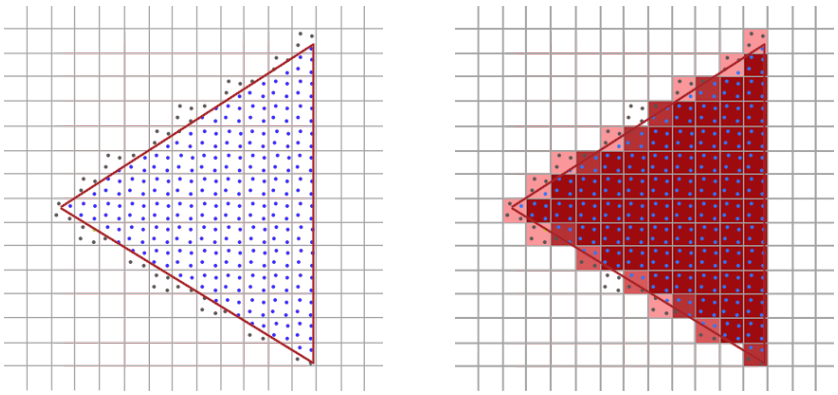
Ο τρόπος που λειτουργεί το ray casting είναι πετώντας μία ακτίνα για το κάθε pixel. Το pixel θα πάρει το χρώμα του αντικειμένου του οποίου χτύπησε. Το πρόβλημα είναι ότι μέσα σε ένα pixel μπορεί να βρίσκονται κομμάτια από παραπάνω από ένα αντικείμενο. Οπότε σε αυτήν την περίπτωση ουσιαστικά θα επιστραφεί ελλιπής πληροφορία. Στην παρακάτω εικόνα απεικονίζεται το πρόβλημα αυτό.



(a) AA rasterization (b) AA rasterization filled

Σχήμα 3.27: Αποτέλεσμα μίας ακτίνας ανά pixel από Joey de Vries *(c) copyright June 2014*

Παρά το γεγονός ότι οι γραμμές του τριγώνου περνάνε μέσα από ένα pixel, άμα δεν περνάει από το κέντρο του τότε θα επιστραφεί το χρώμα του background. Όπως φαίνεται και από την εικόνα 3.23 υπάρχουν διαφορετικοί τρόποι για να γίνει το anti aliasing. Άλλοι με περισσότερα δείγματα και άλλοι με τον ίδιο αριθμό δειγμάτων αλλά σε διαφορετική στοίχιση. Η εργασία αυτή χρησιμοποιεί την 2×2 grid τεχνική.



(a) AA rasterization samples

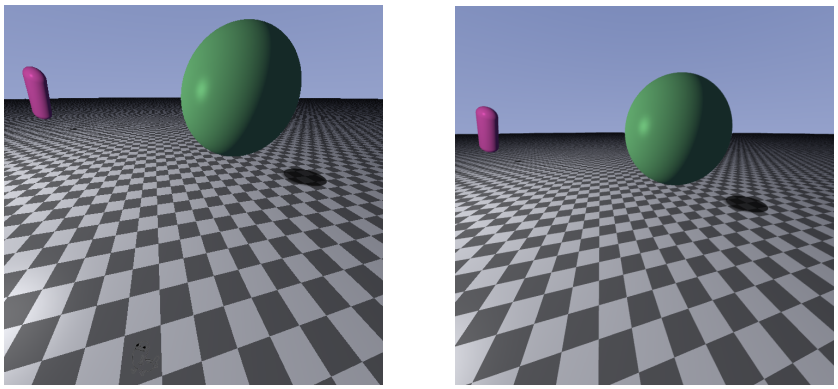
(b) AA rasterization samples filled

Σχήμα 3.28: Αποτέλεσμα 2×2 RGSS τεχνικής από Joey de Vries *(c) copyright June 2014*

Παραπάνω φαίνεται η διαφορά όταν χρησιμοποιηθούν 4 ακτίνες για το κάθε pixel. Η κάθε ακτίνα επιστρέφει ένα 25% από το αντικείμενο που χτύπησε και το τελικό αποτέλεσμα είναι ο μέσος όρος και των τεσσάρων χρωμάτων. Όπως φαίνεται και στην εικόνα 3.23, τα αποτελέσματα είναι διαφορετικά ακόμα και όταν οι ακτίνες στέλνονται από διαφορετικό σημείο στο pixel, ασχέτως άμα είναι ίσες στο πλήθος.

```
uv.x += 0.25 / dims.x;
uv.y += 0.25 / dims.y;
Ray r = castRay(uv);
pixel = vec4(render(r.origin, r.dir), 1.0);
finalColor += pixel;

uv.x += 0.75 / dims.x;
uv.y += 0.25 / dims.y;
r = castRay(uv);
pixel = vec4(render(r.origin, r.dir), 1.0);
finalColor += pixel;

uv.x += 0.25 / dims.x;
uv.y += 0.75 / dims.y;
r = castRay(uv);
pixel = vec4(render(r.origin, r.dir), 1.0);
finalColor += pixel;

uv.x += 0.75 / dims.x;
uv.y += 0.75 / dims.y;
r = castRay(uv);
pixel = vec4(render(r.origin, r.dir), 1.0);
finalColor += pixel;

finalColor /= 4;
```

Παραπάνω η υλοποίηση της τεχνικής 2×2 grid που χρησιμοποιείται στην εργασία. Χωρίζεται σε 4 κομμάτια, το κάθε ένα και για μία ακτίνα. Η uv είναι οι συντεταγμένες των pixel και η dims είναι η ανάλυση της εικόνας. Δημιουργεί offset στον x και στον y 0,25 ώστε να στείλει pixel στην πάνω αριστερή γωνία. Έπειτα γίνονται οι υπολογισμοί της ακτίνας και προσθέτει το χρώμα στην finalColor. Αυτό γίνεται και για τις άλλες ακτίνες αλλά με διαφορετικό offset. Στο τέλος διαιρείται το χρώμα αύτο με το πλήθος των ακτίνων για να μην βγουν "καμμένα" τα χρώματα.



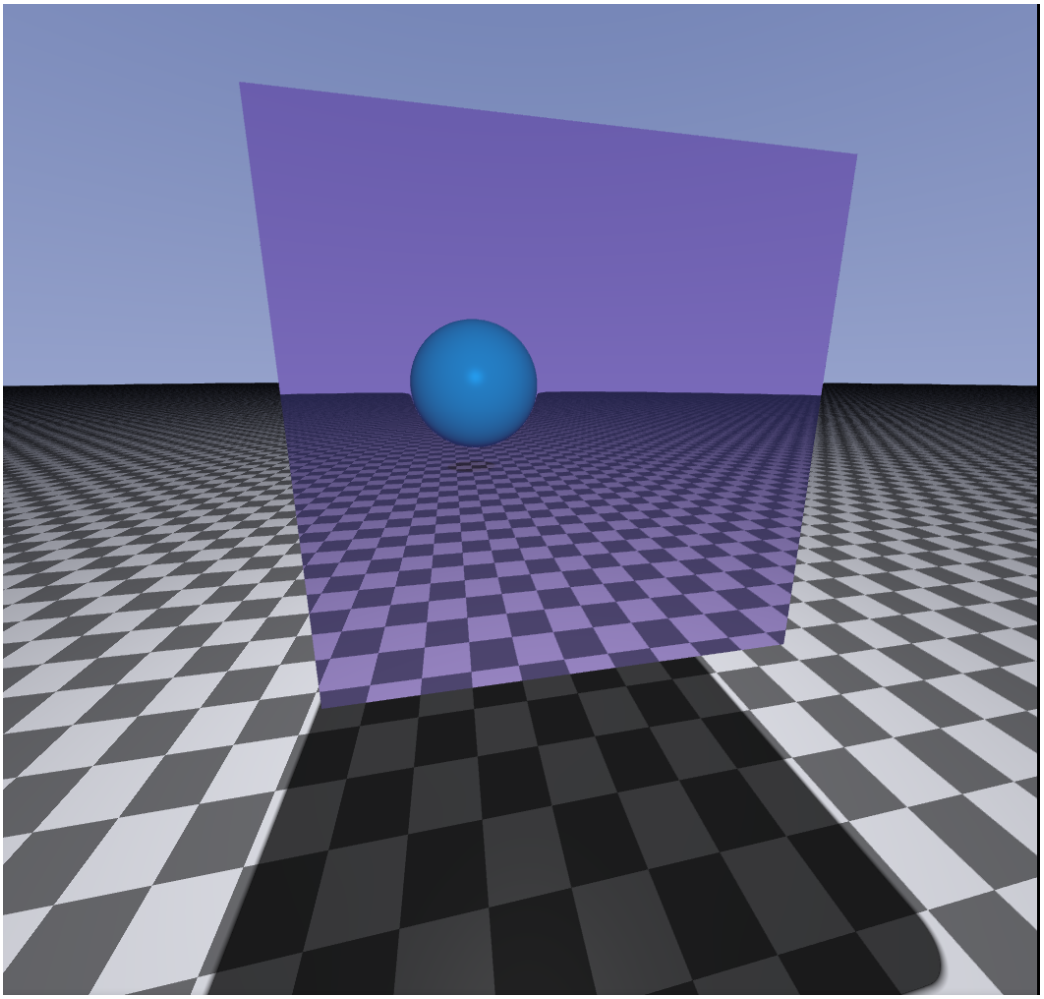
(a) Anti Aliasing Off (b) Anti Aliasing On filled

Σχήμα 3.29: 4x multi-sample anti aliasing

Χρησιμοποιώντας 4x multi-sample anti aliasing γίνεται η ίδια δουλειά που γινόταν και πριν, αλλά 4 φορές. Αυτό σημαίνει ότι και τα καρέ που θα παράγονται θα είναι 4 φορές λιγότερα. Συνήθως στα video games υπάρχουν αρκετές τεχνικές anti aliasing για να επιλέξει ο χρήστης, συνήθως απο 2xMSAA εώς και 8xMSAA. Δίνονται σαν επιλογή και κάποιες κάκιστες τεχνικές όπως το fast approximate anti-aliasing που απλά θολώνουν τα πάντα.

3.10 Αντανακλάσεις

Οι αντανακλάσεις είναι πολύ βασικές στην δημιουργία φωτορεαλιστικών γραφικών. Χάρη στις αντανακλάσεις μπορεί να δοθεί η αίσθηση διαφορετικών υλικών, όπως matte υφές ή αντανακλαστικές. Στον πραγματικό κόσμο οι αναπηδήσεις που μπορεί να κάνει μία ακτίνα φωτός μπορεί να είναι πάρα πολλές μέχρι να εξασθενήσει, ωστόσο επειδή αυτό θα ήταν πολύ σπάταλο στους υπολογιστές, ορίζεται ένα όριο στις αναπηδήσεις της ακτίνας. Το όριο στην εργασία αυτή είναι 5, αν και μετά την τρίτη αντανάκλαση το αποτέλεσμα δεν είναι και πολύ αντιληπτό.



Σχήμα 3.30: Αντανάκλαση με μία αναπήδηση

Στην εικόνα 3.30 φαίνεται η αντανάκλαση της σφαίρας από τον κύβο. Υπάρχουν αρκετά στάδια γι'αυτό το αποτέλεσμα. Για αρχή, το πάτωμα δεν ανταναχλά και γι'αυτο τον λόγο οι αρχικές ακτίνες που χτυπάνε στο πάτωμα δεν μπαίνουν στην διαδικασία της αντανάκλασης, απλά ελέγχεται άμα είναι σε σκίαση και επιστρέφει αμέσως το χρώμα. Σε περίπτωση που η αρχική ακτίνα δεν χτυπήσει ουρανό ή το πάτωμα τότε καλείται η συνάρτηση για την αναπήδηση της ακτίνας, εφόσον επιλεχθεί από τον χρήστη ο αριθμός αναπηδήσεων σε πραγματικό χρόνο.

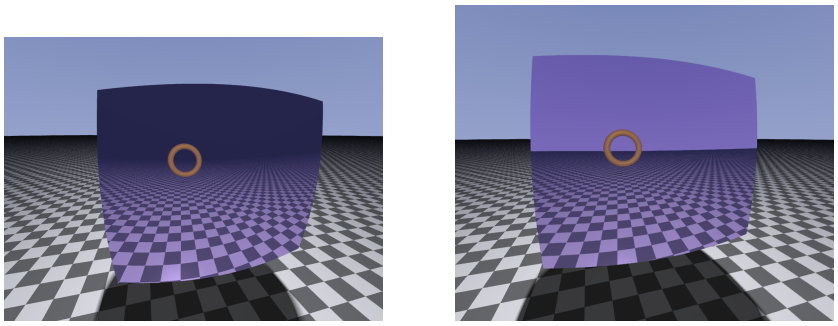
```
vec3 bounce(vec3 rayDir, vec3 pos, vec3 normal, vec3 color, RayHit
    primaryObject)
{
    RayHit prevObject = primaryObject;
    vec3 prevColor = primaryObject.color;
    float shadow = 1.0;

    for (int i = 1; i <= bounceVar; i++)
    {
        rayDir              = reflect(rayDir, normal);
        RayHit t            = reflectedRay(pos + normal * 0.001, rayDir);
        pos                 = pos + rayDir * t.hitpoint;
        normal              = GetNormal(pos);

        if (t.hitpoint == -1.0)
            t.color             = vec3(0.36,0.36,0.60) - (rayDir.y * 0.2);
        else
            t.color         = getPointLight(t.color, normal, pos);

        if (t.id == 7 && prevObject.material != MATTE && i < 3)
        {
            vec3 shadowRayOrigin = pos + normal * 0.02;
            vec3 shadowRayDir = light.position - pos;
            shadow = softshadow(shadowRayOrigin, shadowRayDir, 2.0);
            color *= shadow / i;
        }

        if (prevObject.material == MATTE)
            continue;
        else
            color          += t.color * prevColor / i;

        prevColor           = t.color;
        prevObject          = t;
    }

    return color;
}
```

Όταν καλείται η συνάρτηση στέλνονται μαζί της όλα τα χαρακτηριστικά του αντικειμένου όπου ξεκινάει η αντανάκλαση.

Ο τύπος **RayHit** κρατάει κάποια δεδομένα για το αντικείμενο. Το σημείο που χτύπησε, το χρώμα του αντικειμένου, τον αριθμό χαρακτηριστικού (0 έως n) και το υλικό του όπου θα είναι είτε matte ή reflective.

```
struct RayHit {
    float hitpoint;
    vec3 color;
    int id;
    float material;
};
```

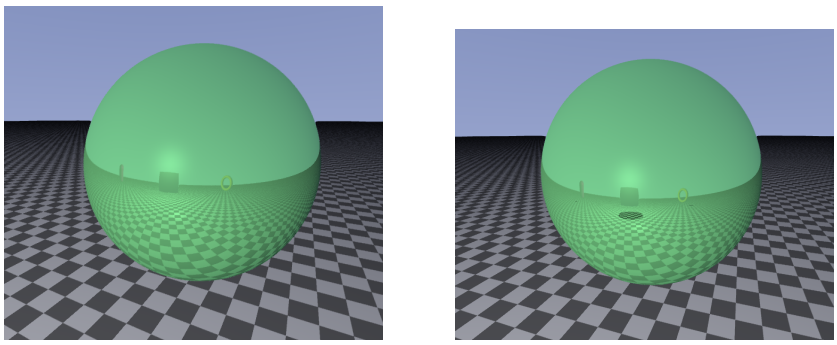
Το αρχικό αντικείμενο αποθηκεύεται σαν το προηγούμενο. Έπειτα ξεκινάει η λούπα για τις αναπηδήσεις. Η συνάρτηση reflectedRay κάνει ακριβώς ότι και η RayMarch μόνο που χρησιμοποιεί τα μισά βήματα για να είναι πιο βέλτιστος ο αλγόριθμος. Επόμενο βήμα είναι να ελέγξει άμα γίνει αντανάκλαση και η ακτίνα πάει προς τον ουρανό. Σε αυτήν την περίπτωση το αντικείμενο πρέπει να πάρει το χρώμα του ουρανού.



(a) (b)

Σχήμα 3.31: Στην εικόνα (a) παίρνει το χρώμα του αντικειμένου ανάλογα πόσο το χτυπάει το φως ενώ στην (b) παίρνει το χρώμα του ουρανού όπως θα έπρεπε

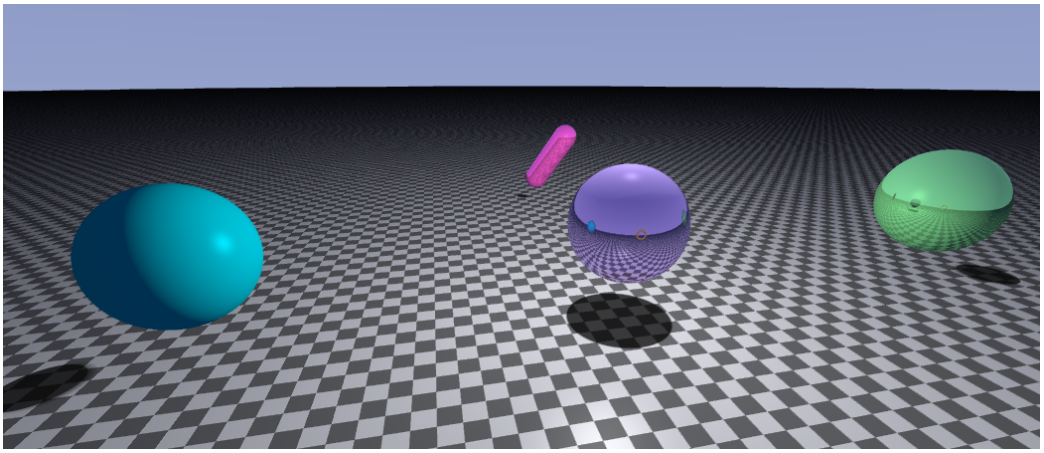
Στην περίπτωση που δεν πάει η ακτίνα προς τον ουρανό πρέπει να υπολογιστεί ξανά η φωτεινότητα από το αντικείμενο που χτύπησε η ακτίνα. Έπειτα πρέπει να ελεγχθεί ξανά άμα η ανταναακλώμενη ακτίνα χτύπησε σε σημείο που είναι σε σκίαση. Ο έλεγχος αυτός θα γίνει μόνο άμα το αντικείμενο αυτό είναι το πάτωμα (δηλαδή το id 7), το προηγούμενο αντικείμενο είναι διαφορετικό του matte και ο έλεγχος να γίνει για τις πρώτες 2 ανταναακλάσεις.



(a) Μη ανταναακλώμενες σκιές (b) Ανταναακλώμενες σκιές

Σχήμα 3.32: Διαφορές ανάμεσα στον έλεγχο σκιάς για την ανταναακλώμενη ακτίνα

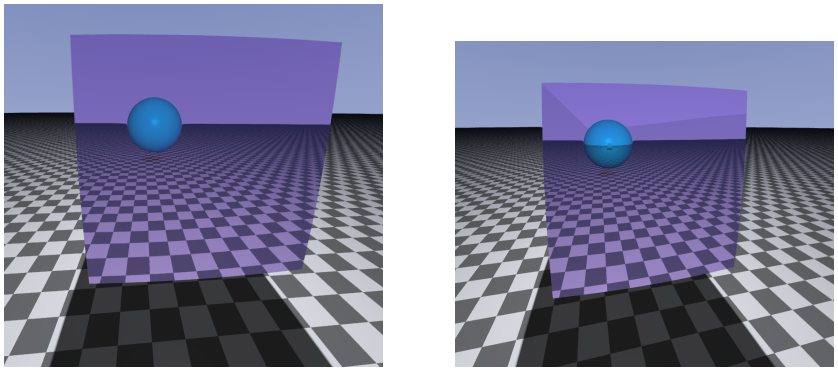
Η διαφορά είναι αρκετά αντιληπτή ακόμα και κοιτάζοντας μόνο την εικόνα (a) κάτι φαίνεται λάθος. Επίσης ελέγχεται άμα το αντικείμενο είναι διαφορετικό του matte, γιατί στην περίπτωση που είναι θα μοιάζει όπως στην εικόνα 3.33.



Σχήμα 3.33: Μη ανταναακλαστικό αντικείμενο

Στην περίπτωση αυτή η λούπα δεν συνεχίζει στις παρακάτω πράξεις και πάει στο επόμενο βήμα. Αλλιώς προστίθεται στο τελικό χρώμα, το χρώμα του νέου αντικειμένου επί ένα ποσοστό από το χρώμα του προηγούμενου. Το ποσοστό αυτό βγαίνει διαιρώντας με τον αριθμό της αναπήδησης της ακτίνας. Όσο πιο μεγάλος ο αριθμός της αναπήδησης τόσο λιγότερο θα επηρεάσει και με το χρώμα της, όπως και στον πραγματικό κόσμο.

Τέλος μερικά παραδείγματα ανάμεσα σε μία αντανάκλαση με περισσότερες.



(a) Μια αντανάκλαση (b) Δύο ανταναακλάσεις

Σχήμα 3.34: Διαφορές ανάμεσα σε μία και δύο αναπηδήσεις της ακτίνας

Όπως φαίνεται οι παραπάνω αναπηδήσεις της ακτίνας δημιουργούν ανταναακλάσεις μέσα στις ανταναακλάσεις. Μετά από ένα σημείο είναι άσκοπο ειδικά άμα η ανάλυση της εικόνας είναι μικρή. Σε video games που χρησιμοποιείται ray tracing συνηθίζεται να υπάρχει μέχρι μία αναπήδηση της ακτίνας εφόσον είναι πολύ σπάταλη διαδικασία, αλλά σε κινούμενα σχέδια που δεν έχει σημασία άμα το rendering θα γίνει γρήγορα τότε υπάρχουν και περισσότερες αναπηδήσεις και μεγαλύτερη λεπτομέρεια στον φωτορεαλισμό. Με λίγα λόγια δεν υπάρχουν τσιγκουνιές όταν δεν είναι πραγματικού χρόνου το rendering.

Κεφάλαιο 4

Συμπεράσματα και Βελτιώσεις

Συνολικά, έχουν υλοποιηθεί τα περισσότερα πράγματα από αυτά που είχαν σκοπό να υλοποιηθούν σε αυτήν την εργασία. Χρησιμοποιώντας την OpenGL η οποία δεν είναι φτιαγμένη για την δημιουργία ενός τέτοιου αλγόριθμου τροποποιήθηκε ώστε να καλύψει τις ανάγκες της εργασίας.

Θέλοντας να γίνει αυτήν η εργασία αυτόνομη, δηλαδή να μην χρειάζεται να τρέχει ο κώδικα για ray marching στο shadertoy, το οποίο ουσιαστικά λειτουργεί με τον ίδιο τρόπο, χρειαζόντουσαν γνώσεις σε OpenGL για το στήσιμο του renderer. Έτσι δίνεται ελευθερία στον χρήστη να κάνει περισσότερα πράγματα με τον αλγόριθμο, όπως να στέλνει στον shader που κάνει το ray marching ότι μεταβλητές θέλει (το shadertoy δίνει κάποιες συγκεκριμένες στον χρήστη όπως τον χρόνο).

Μερικές περαιτέρω υλοποιήσεις στο μέλλον για την εργασία είναι η διάθλαση της ακτίνας, καλύτερος φωτορεαλισμός και πιο ρεαλιστικές σκιές, περίπλοκα σχήματα ενώνοντας πολλά μαζί και διάφορες τεχνικές βελτιστοποίησης του αλγορίθμου όπως η boundary volume hierarchy.

Συμπερασματικά, η εργασία αναπαριστά δυνατότητες του ray marching και τι μπορεί να δημιουργηθεί μέσα σε έναν shader. Ο κώδικας είναι γραμμένος για linux, αλλά με μερικές αλλαγές γίνεται να τρέξει και στα windows.

Παράρτημα Α

Παράρτημα κώδικα

Το project βρίσκεται σε repository στο github προφίλ μου μαζί με οδηγίες εγκατάστασης. Link: **Ray Marching in OpenGL**

A.1 main.cpp

```
#define GL_GLEXT_PROTOTYPES
#include "includes/glm/glm.hpp"
#include "includes/glm/gtc/matrix_transform.hpp"
#include "includes/glm/gtc/type_ptr.hpp"

#include "source/shader.hpp"
#include "source/camera.cpp"
#include "source/MousePosition.cpp"
#include "source/quad.cpp"
#include "source/texture.cpp"
#include "source/WorkGroups.hpp"

#include <GLFW/glfw3.h>
#include <iostream>

const unsigned int SCREEN_WIDTH  = 1080;
const unsigned int SCREEN_HEIGHT = 1080;

void processInput(GLFWwindow *window);
void mouse_callback(GLFWwindow *window, double xpos, double ypos);
void key_callback(GLFWwindow* window, int key, int scancode, int action, int
    mods);

bool zaxisPos   = false;
bool zaxisNeg   = false;
bool xaxisPos   = false;
bool xaxisNeg   = false;
bool AA         = true;
bool showQuad   = false;
```

```
float halfSpeed = false;
int bounce = 0;

float deltaTime = 0.0f;
float lastFrame = 0.0f;

float lastX     = SCREEN_WIDTH / 2.0;
float lastY     = SCREEN_HEIGHT / 2.0;
bool firstMouse = true;

MouseInput mouse;
Camera camera(SCREEN_WIDTH, SCREEN_HEIGHT, 0.025, 10.0, glm::vec3(0, 0, 0),
    glm::vec3(0, 0, -1), glm::vec3(0, 1, 0));

int main(void)
{
    GLFWwindow *window;

    glfwInit();

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Ray Marching",
        nullptr, nullptr);
    if (!window) {
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    glfwSetKeyCallback(window, key_callback);

    // Create quad
    FullScreenQuad quad;
    quad.GenBuffer();


    // create texture for compute shader
    Texture tex(SCREEN_WIDTH, SCREEN_HEIGHT);
    tex.GenerateTexture();


    // printWorkGroupCount();
    // printWorkGroupSize();
    unsigned int invocations = printInvocations();
    unsigned int workgroups = sqrt(invocations);
    std::cout << "Maximum Workgroups:\t" << workgroups << std::endl;

    // Quad's Fragment and Vertex shader
    ShaderProgramSource basic = ParseShader("shaders/Quad.glsl");
    unsigned int quadShader   = CreateShader(basic.VertexShader, basic.
        FragmentShader);

    // Compute Shader
    const std::string &compShader = ParseCompute("shaders/computeShader.glsl
        ");
    unsigned int marching         = CreateCompute(compShader);
```

```
double lastTime      = 0.0;
unsigned int counter = 0;

// Game loop
while (!glfwWindowShouldClose(window)) {
    float currentFrame = glfwGetTime();
    deltaTime          = currentFrame - lastFrame;
    lastFrame          = currentFrame;

    processInput(window);

    useShader(marching);
    setFloat(marching, "iTime", (float)glfwGetTime());
    setuInt(marching, "workgroups", &workgroups);

    setVec4(marching, "camera.pos", camera.cameraPos.x, camera.cameraPos
        .y, camera.cameraPos.z, 0.0);
    setVec4(marching, "camera.dir", camera.forward.x, camera.forward.y,
        camera.forward.z, 0.0);
    setVec4(marching, "camera.yAxis", camera.up.x, camera.up.y, camera.
        up.z, 0.0);
    setVec4(marching, "camera.xAxis", camera.right.x, camera.right.y,
        camera.right.z, 0.0);

    setVec3(marching, "light.position", glm::vec3(-5, 5, -10));
    setVec3(marching, "light.ambient", glm::vec3(0.03, 0.04, 0.1));
    setVec3(marching, "light.diffuse", glm::vec3(0.8, 0.8, 0.8));
    setVec3(marching, "light.specular", glm::vec3(0.5, 0.5, 0.5));
    setFloat(marching, "light.constant", 1.0);
    setFloat(marching, "light.linear", 0.009);
    setFloat(marching, "light.quadratic", 0.00032);

    setBool(marching, "AA", AA);
    setInt(marching, "bounceVar", bounce);
    setFloat(marching, "drand48", drand48());
    setVec3(marching, "mouse", mouse.EulerAngles());
    setVec2(marching, "iMouse", glm::vec2(mouse.yaw, mouse.pitch));

    // Number of work groups in dispach: X, Y, Z
    glDispatchCompute(SCREEN_WIDTH / workgroups, SCREEN_HEIGHT /
        workgroups, 1);

    glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);

    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    useShader(quadShader);

    glBindVertexArray(quad.quadVAO);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, tex.texOutput);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    ++counter;
    double currentTime = glfwGetTime();

    if (currentTime - lastTime >= 1.0) {
        std::cout << "FPS:\t" << counter << std::endl;
        ++lastTime;
        counter = 0;
    }
```

```
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    quad.DeleteVertex();
    quad.DeleteBuffer();

    glfwTerminate();
    return 0;
}

void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        zaxisNeg = true;
    else {
        zaxisNeg = false;
    }

    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        zaxisPos = true;
    else {
        zaxisPos = false;
    }

    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        xaxisPos = true;
    else {
        xaxisPos = false;
    }

    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        xaxisNeg = true;
    else {
        xaxisNeg = false;
    }

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS && glfwGetKey(window,
        GLFW_KEY_A) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_D) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_S) == GLFW_PRESS ||
        glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS && glfwGetKey(window,
            GLFW_KEY_D) == GLFW_PRESS) {
        halfSpeed = true;
    }
    else {
        halfSpeed = false;
    }

    camera.lookAt(zaxisNeg, zaxisPos, xaxisNeg, xaxisPos, halfSpeed,
        deltaTime);
}

void key_callback(GLFWwindow* window, int key, int scancode, int action, int
     mods)
{
    if (key == GLFW_KEY_UP && action == GLFW_PRESS)
```

```
        if (bounce < 5)
            bounce += 1;
    if (key == GLFW_KEY_DOWN && action == GLFW_PRESS)
        if (bounce > 0)
            bounce -= 1;

    if (key == GLFW_KEY_F1 && action == GLFW_PRESS)
        AA = !AA;

    if (key == GLFW_KEY_L && action == GLFW_PRESS)
    {
        if (showQuad)
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        else
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        showQuad = !showQuad;
    }
}

void mouse_callback(GLFWwindow *window, double xpos, double ypos)
{
    if (firstMouse) {
        lastX      = xpos;
        lastY      = ypos;
        firstMouse = false;
    }

    float xoffset = lastX - xpos;
    float yoffset = lastY - ypos;
    lastX         = xpos;
    lastY         = ypos;

    mouse.ProcessMouseOffset(xoffset, yoffset);
    camera.setMouse(-xpos, -ypos);
}
// g++ main.cpp -lGL -lglfw && ./a.out
```

A.2 Mouseposition.hpp και Mouseposition.cpp

```
#pragma once
#include "../includes/glm/glm.hpp"
#include "../includes/glm/gtc/matrix_transform.hpp"
#include "../includes/glm/gtc/type_ptr.hpp"
#include <GLFW/glfw3.h>
#include <iostream>

float MouseSensitivity = 0.001;

class MouseInput {
  public:
    float pitch, yaw;

    MouseInput() noexcept;

    MouseInput(const float newPitch, const float newYaw) noexcept;

    void ProcessMouseOffset(float xoffset, float yoffset);

    glm::vec3 EulerAngles(); //You can use this for color input, it's fun I
        guess
};
```

```
#include <iostream>
#include "MousePosition.hpp"

MouseInput::MouseInput() noexcept
: pitch(0.0), yaw(0.0) {}

MouseInput::MouseInput(const float newPitch, const float newYaw) noexcept
: pitch(newPitch), yaw(newYaw) {}

void MouseInput::ProcessMouseOffset(float xoffset, float yoffset)
{
    xoffset *= MouseSensitivity;
    yoffset *= MouseSensitivity;

    yaw += xoffset;
    pitch += yoffset;

    // if (pitch > 89.0)
    //     pitch = 89.0;

    // if (pitch < -89.0)
    //     pitch = -89.0;
}

glm::vec3 MouseInput::EulerAngles()
{
    glm::vec3 front;
    front.x = cos(glm::radians(yaw) * cos(glm::radians(pitch)));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw) * cos(glm::radians(pitch)));

    return glm::vec3(glm::normalize(front));
}
```

A.3 Workgroups

```
#include <iostream>
#define GL_GLEXT_PROTOTYPES
#include <GLFW/glfw3.h>

void printWorkGroupCount()
{
    int work_grp_cnt[3];

    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 0, &work_grp_cnt[0]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 1, &work_grp_cnt[1]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 2, &work_grp_cnt[2]);

    std::cout << "max global (total) work group size x: "  << work_grp_cnt
        [0]
                                                  << " y: " << work_grp_cnt
                                                      [1]
                                                  << " z: " << work_grp_cnt
                                                      [2] << std::endl << std
                                                      ::endl;
}

void printWorkGroupSize()
{
    int work_grp_size[3];

    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 0, &work_grp_size[0]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 1, &work_grp_size[1]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 2, &work_grp_size[2]);

    std::cout << "max local (in order) work group sizes x: "  <<
        work_grp_size[0]
                                                     << " y: " <<
                                                         work_grp_size[1]
                                                     << " z: " <<
                                                         work_grp_size[2] <<
                                                         std::endl << std::
                                                         endl;
}

int printInvocations()
{
    int work_grp_inv;
    glGetIntegerv(GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS, &work_grp_inv);

    std::cout << "max local work group invocations " << work_grp_inv << std
        ::endl << std::endl;

    return work_grp_inv;
}
```

Α.4 camera.hpp και camera.cpp

```
#pragma once
#include "../includes/glm/glm.hpp"
#include "../includes/glm/gtc/matrix_transform.hpp"
#include "../includes/glm/gtc/type_ptr.hpp"
#include <GLFW/glfw3.h>
#include <iostream>

typedef glm::vec3 vec3;
typedef glm::vec4 vec4;
typedef glm::mat4 mat4;

class Camera {
  public:
    int width;
    int height;
    float angleY;
    float angleX;
    float mouseSensitivity;
    float keyboardSpeed;
    float xpos;
    float ypos;

    vec3 cameraPos;
    vec3 forward;
    vec3 up;
    vec3 right;

    Camera() noexcept;

    Camera(int width, int height, float mouseSensitivity, float
        keyboardSpeed, vec3 pos, vec3 lookAt, vec3 up) noexcept;

    void setMouse(float x, float y);

    void lookAt(bool zN, bool zP, bool xN, bool xP, bool halfSpeed, float
        deltaTime);
};
```

```
#include <iostream>
#include "camera.hpp"

Camera::Camera() noexcept
: width(1024), height(1024), angleY(0.0), angleX(0.0), mouseSensitivity(1.0)
    , keyboardSpeed(10.0),
          cameraPos(vec3(0)), forward(vec3(0, 0, -1)), up(vec3(0, 1, 0)),
              right(vec3(1, 0, 0)) {}

Camera::Camera(int width, int height, float mouseSensitivity, float
    keyboardSpeed, vec3 pos, vec3 lookAt, vec3 up) noexcept
: width(width), height(height), mouseSensitivity(mouseSensitivity),
    cameraPos(pos), keyboardSpeed(keyboardSpeed)
{
    forward = glm::normalize(lookAt - pos);
    right   = glm::normalize(glm::cross(up, forward));
    up      = glm::cross(right, forward);
}

void Camera::setMouse(float x, float y)
```

```
{
    xpos = x;
    ypos = y;
}

void Camera::lookAt(bool zN, bool zP, bool xN, bool xP, bool halfSpeed,
    float deltaTime)
{
    angleX = xpos * mouseSensitivity;
    angleY = ypos * mouseSensitivity;

    mat4 rotateX = glm::rotate(mat4(1.0), glm::radians(angleY), vec3(1.0,
        0.0, 0.0));
    mat4 rotateY = glm::rotate(mat4(1.0), glm::radians(angleX), vec3(0.0,
        1.0, 0.0));

    mat4 rotationMatrix = rotateY * rotateX;

    // convert the mat4 to vec4 and get the the xyz
    forward = glm::normalize(vec3(rotationMatrix * vec4(0.0, 0.0, -1.0, 0.0)
        ));
    up      = glm::normalize(vec3(rotationMatrix * vec4(0.0, 1.0, 0.0, 0.0))
        );
    right   = glm::normalize(glm::cross(forward, up));

    if (halfSpeed) {
        keyboardSpeed = 5.0;
    }
    else
        keyboardSpeed = 10.0;

    if (zN) // W
        cameraPos += (keyboardSpeed * forward) * deltaTime;
    if (zP) // S
        cameraPos += (keyboardSpeed * (-forward)) * deltaTime;
    if (xN) // A
        cameraPos += (keyboardSpeed * (-right)) * deltaTime;
    if (xP) // D
        cameraPos += (keyboardSpeed * right) * deltaTime;
}
```

A.5 quad.hpp και quad.cpp

```
#include <iostream>

class FullScreenQuad
{
public:
    float quadVertices[24] = {
            //positions   texture Coords
            -1.0f,  1.0f, 0.0f, 1.0f,
            -1.0f, -1.0f, 0.0f, 0.0f,
             1.0f, -1.0f, 1.0f, 0.0f,

            -1.0f,  1.0f, 0.0f, 1.0f,
             1.0f, -1.0f, 1.0f, 0.0f,
             1.0f,  1.0f, 1.0f, 1.0f
        };
    unsigned int quadVAO, quadVBO;

    void GenBuffer();
    void DeleteVertex();
    void DeleteBuffer();
};
```

```
#include <iostream>
#include "quad.hpp"

void FullScreenQuad::GenBuffer()
{
    glGenVertexArrays(1, &quadVAO);
    glGenBuffers(1, &quadVBO);
    glBindVertexArray(quadVAO);
    glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &quadVertices,
        GL_STATIC_DRAW);

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void
         *)0);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void
         *)(2 * sizeof(float)));
    glEnableVertexAttribArray(1);
}

void FullScreenQuad::DeleteVertex()
{
    glDeleteVertexArrays(1, &quadVAO);
}

void FullScreenQuad::DeleteBuffer()
{
    glDeleteBuffers(1, &quadVBO);
}
```

A.6 shader.hpp

```
#pragma once
#define GL_GLEXT_PROTOTYPES
#include "../includes/glm/glm.hpp"
#include "../includes/glm/gtc/matrix_transform.hpp"
#include "../includes/glm/gtc/type_ptr.hpp"
#include <GLFW/glfw3.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

struct ShaderProgramSource {
    std::string VertexShader;
    std::string FragmentShader;
};

void useShader(unsigned int program) { glUseProgram(program); }

void setBool(unsigned int ID, const std::string &name, bool value)
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}

void setInt(unsigned int ID, const std::string &name, int value)
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}

void setuInt(unsigned int ID, const std::string &name, unsigned int *value)
{
    glUniform1uiv(glGetUniformLocation(ID, name.c_str()), 1, value);
}

void setFloat(unsigned int ID, const std::string &name, float value)
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}

void setVec2(unsigned int ID, const std::string &name, const glm::vec2 &
    value)
{
    glUniform2fv(glGetUniformLocation(ID, name.c_str()), 1, &value[ 0 ]);
}

void setVec2(unsigned int ID, const std::string &name, float x, float y)
{
    glUniform2f(glGetUniformLocation(ID, name.c_str()), x, y);
}

// ------------------------------------------------------------------------
void setVec3(unsigned int ID, const std::string &name, const glm::vec3 &
    value)
{
    glUniform3fv(glGetUniformLocation(ID, name.c_str()), 1, &value[ 0 ]);
}

void setVec3(unsigned int ID, const std::string &name, float x, float y,
    float z)
{
```

```
    glUniform3f(glGetUniformLocation(ID, name.c_str()), x, y, z);
}

// ------------------------------------------------------------------------
void setVec4(unsigned int ID, const std::string &name, const glm::vec4 &
    value)
{
    glUniform4fv(glGetUniformLocation(ID, name.c_str()), 1, &value[ 0 ]);
}

void setVec4(unsigned int ID, const std::string &name, float x, float y,
    float z, float w)
{
    glUniform4f(glGetUniformLocation(ID, name.c_str()), x, y, z, w);
}

// ------------------------------------------------------------------------
void setMat2(unsigned int ID, const std::string &name, const glm::mat2 &mat)
{
    glUniformMatrix2fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
        &mat[ 0 ][ 0 ]);
}

// ------------------------------------------------------------------------
void setMat3(unsigned int ID, const std::string &name, const glm::mat3 &mat)
{
    glUniformMatrix3fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
        &mat[ 0 ][ 0 ]);
}

// ------------------------------------------------------------------------
void setMat4(unsigned int ID, const std::string &name, const glm::mat4 &mat)
{
    glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
        &mat[ 0 ][ 0 ]);
}

static ShaderProgramSource ParseShader(const std::string &filepath)
{
    std::ifstream stream(filepath);
    enum class ShaderType { NONE = -1, VERTEX = 0, FRAGMENT = 1 };

    std::string line;
    std::stringstream ss[ 2 ];
    ShaderType type = ShaderType::NONE;
    while (getline(stream, line)) {
        if (line.find("#shader") != std::string::npos) {
            if (line.find("vertex") != std::string::npos)
                type = ShaderType::VERTEX;
            else if (line.find("fragment") != std::string::npos)
                type = ShaderType::FRAGMENT;
        }
        else
            ss[ (int)type ] << line << '\n';
    }
    return {ss[ 0 ].str(), ss[ 1 ].str()};
}

static unsigned int CompileShader(unsigned int type, const std::string &
    source)
{
    unsigned int id = glCreateShader(type);
```

```
    const char *src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE) {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char message[ 512 ];
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile" << (type == GL_VERTEX_SHADER ? "
            vertex" : " fragment") << " shader!"
                << std::endl;
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }
    return id;
}

static unsigned int CreateShader(const std::string &VertexShader, const std
    ::string &FramgneShader)
{
    unsigned int program = glCreateProgram();
    unsigned int vs      = CompileShader(GL_VERTEX_SHADER, VertexShader);
    unsigned int fs      = CompileShader(GL_FRAGMENT_SHADER, FramgneShader);
    glAttachShader(program, vs);
    glAttachShader(program, fs);
    glLinkProgram(program);
    glValidateProgram(program);

    glDeleteShader(vs);
    glDeleteShader(fs);

    return program;
}

static std::string ParseCompute(const std::string &filepath)
{
    std::ifstream stream(filepath);

    enum class ShaderType { NONE = -1, COMPUTE = 0 };

    std::string line;
    std::stringstream ss;
    ShaderType type = ShaderType::NONE;
    while (getline(stream, line)) {
        ss << line << '\n';
    }
    return {ss.str()};
}

static unsigned int CompileCompute(unsigned int type, const std::string &
    source)
{
    unsigned int id = glCreateShader(type);
    const char *src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
```

```
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE) {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char message[ 512 ];
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile compute shader!" << std::endl;
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }
    return id;
}

static unsigned int CreateCompute(const std::string &ComputeShader)
{
    unsigned int program = glCreateProgram();
    unsigned int cs      = CompileCompute(GL_COMPUTE_SHADER, ComputeShader);
    glAttachShader(program, cs);
    glLinkProgram(program);
    glValidateProgram(program);

    glDeleteShader(cs);

    return program;
}
```

A.7 texture.hpp και texture.cpp

```
#include <iostream>

class Texture
{
public:
    int texWidth;
    int texHeight;
    unsigned int texOutput;

    Texture() noexcept;
    Texture(const unsigned int SCREEN_WIDTH, const unsigned int
        SCREEN_HEIGHT) noexcept;

    void GenerateTexture();
};
```

```
#include <iostream>
#include "texture.hpp"

Texture::Texture() noexcept
: texWidth(720), texHeight(720) {}

Texture::Texture(const unsigned int SCREEN_WIDTH, const unsigned int
    SCREEN_HEIGHT) noexcept
: texWidth(SCREEN_WIDTH), texHeight(SCREEN_HEIGHT) {}

void Texture::GenerateTexture()
{
    glGenTextures(1, &texOutput);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texOutput);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, texWidth, texHeight, 0,
        GL_RGBA, GL_FLOAT, nullptr);
    glBindImageTexture(0, texOutput, 0, GL_FALSE, 0, GL_WRITE_ONLY,
        GL_RGBA32F);
}
```

A.8 Quad.glsl

```
#shader vertex
#version 330 core
layout(location = 0) in vec2 aPos;
layout(location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    TexCoords   = aTexCoords;
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}

#shader fragment
#version 330 core

in vec2 TexCoords;
out vec4 color;
uniform sampler2D screenTexture;

void main()
{
    vec3 col     = texture(screenTexture, TexCoords).rgb;
    color = vec4(col, 1.0);
}
```

A.9 computeShader.glsl

```
#version 450
#define MAX_STEPS 512
#define MIN_DIST .000001
#define REFLECTIVE 1.0
#define MATTE      0.0

uniform uint workgroups;

const uint TILE_W      = workgroups;
const uint TILE_H      = workgroups;
const ivec2 TILE_SIZE = ivec2(TILE_W, TILE_H);
layout(local_size_x = 39, local_size_y = 39) in;
layout(rgba32f, binding = 0) uniform image2D img_output;

struct Camera {
    vec4 pos;
    vec4 dir;
    vec4 yAxis;
    vec4 xAxis;
};

struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};

struct Ray {
    vec3 origin;
    vec3 dir;
};

struct RayHit {
    float hitpoint;
    vec3 color;
    int id;
    float material;
};

// SDFs and Soft shadows by Inigo quilez https://www.iquilezles.org/www/
    index.htm
float sdSphere(vec3 p, float r);
RayHit sdf(vec3 pos);
RayHit RayMarch(vec3 rayOrigin, vec3 rayDir);
RayHit reflectedRay(vec3 rayOrigin, vec3 rayDir);
vec3 render(vec3 rayOrigin, vec3 rayDir);
vec3 GetNormal(vec3 pos);
float sdPlane(vec3 p, vec4 n);
vec3 getPointLight(vec3 color, vec3 normal, vec3 pos); // https://
    learnopengl.com/Lighting/Light-casters
vec3 bounce(vec3 rayDir, vec3 pos, vec3 normal, vec3 color, RayHit
    primaryObject);
float softshadow(vec3 ro, vec3 rd, float k);
```

```
vec3 checkers(vec3 p);

uniform bool AA;
uniform int bounceVar; //number of bouncing rays
uniform float drand48; // testing
uniform float iTime; // Time since glfw was initialized
uniform vec3 mouse; // Calculated Euler Angles
uniform vec2 iMouse; // Carries the current pixel location of the mouse
    cursor
uniform Camera camera; // Camera rotation and movement values
uniform Light light;

Ray castRay(vec2 uv)
{
    float Perpective = radians(45);
    vec4 dir         = normalize(uv.x * camera.xAxis + uv.y * camera.yAxis +
         camera.dir * Perpective);

    return Ray(camera.pos.xyz, dir.xyz);
}

// https://www.shadertoy.com/view/3df3DH
vec3 checkers(vec3 p)
{
    return int(1000.0+p.x) % 2 != int(1000.0+p.z) % 2 ? vec3(1.0) /*vec3(
        mouse) * 10*/ : vec3(0.2);
}


float sdSphere(vec3 p, float r) { return length(p) - r; }

float sdPlane(vec3 p, vec4 n) { return dot(p, n.xyz) + n.w; }

float sdBox(vec3 p, vec3 b)
{
    vec3 d = abs(p) - b;
    return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
}

float sdTorus(vec3 p, vec2 t)
{
    return length( vec2(length(p.xz)-t.x,p.y) )-t.y;
}

float sdCapsule(vec3 p, vec3 a, vec3 b, float r)
{
        vec3 pa = p-a, ba = b-a;
        float h = clamp( dot(pa,ba)/dot(ba,ba), 0.0, 1.0 );
        return length( pa - ba*h ) - r;
}

RayHit opU(RayHit d1, RayHit d2) { return (d1.hitpoint < d2.hitpoint) ? d1 :
     d2; }

RayHit sdf(vec3 pos)
{
    // The last vec3 refers to the color of each object
    RayHit t;
    t         = RayHit(sdSphere(pos - vec3(15.0, 0.0, -10.0), 3.0), vec3
        (0.1804, 0.6, 0.2157), 0, REFLECTIVE);
    t         = opU(t, RayHit(sdSphere(pos - vec3(-25.0, 0.0, -10.0), 3.0),
        vec3(0.0, 0.851, 1.0), 1, REFLECTIVE));
```

```
    // Blended shapes
    RayHit Box    = RayHit(sdBox(pos - vec3(-5.0, 0.0, -10.0), vec3
        (3,2.5,2.5)), vec3(1.0, 1.0, 1.0), 2, REFLECTIVE);
    RayHit Sphere = RayHit(sdSphere(pos - vec3(-5.0, 0.0, -10.0), 3.0), vec3
        (1.0, 1.0, 1.0), 3, REFLECTIVE);
    t         = opU(t, RayHit(mix(Box.hitpoint, Sphere.hitpoint, sin(iTime)
        / 2 + 0.5), vec3(0.4863, 0.3529, 0.702), 4, REFLECTIVE));

    t         = opU(t, RayHit(sdTorus((pos - vec3(-5.0, 0.0, 10.0)).xzy,
        vec2(2.5, 0.5)), vec3(0.9137, 0.549, 0.0), 5, REFLECTIVE));
    t         = opU(t, RayHit(sdCapsule(pos - vec3(-5.0, -2.0, -30.0), vec3
        (-0.1,0.1,-0.1), vec3(2.0,4.0,2.0), 1.0), vec3(0.8, 0.0902, 0.4824),
         6, REFLECTIVE));
    t         = opU(t, RayHit(sdPlane(pos, vec4(0, 1, 0, 5.5)), checkers(pos
        ), 7, MATTE));
    return t;
}

RayHit RayMarch(vec3 rayOrigin, vec3 rayDir)
{
    float t = 0.0; // Stores current distance along ray
    float tmax = 400;
    RayHit dummy = {-1.0, vec3(0.0), -1, 1.0};

    for (int i = 0; i < MAX_STEPS; i++) {
        RayHit res = sdf(rayOrigin + rayDir * t);
        if (res.hitpoint < (MIN_DIST * t)) {
            return RayHit (t, res.color, res.id, res.material);
        }
        if (res.hitpoint > tmax)
            return dummy;
        t += res.hitpoint;
    }

    return dummy;
}

RayHit reflectedRay(vec3 rayOrigin, vec3 rayDir)
{
    float t = 0.0; // Stores current distance along ray
    float tmax = 200;
    RayHit dummy = {-1.0, vec3(0.0), -1, 1.0};

    for (int i = 0; i < MAX_STEPS / 2; i++) {
        RayHit res = sdf(rayOrigin + rayDir * t);
        if (res.hitpoint < (MIN_DIST * t)) {
            return RayHit (t, res.color, res.id, res.material);
        }
        if (res.hitpoint > tmax)
            return dummy;
        t += res.hitpoint;
    }

    return dummy;
}

vec3 bounce(vec3 rayDir, vec3 pos, vec3 normal, vec3 color, RayHit
    primaryObject)
{
    RayHit prevObject = primaryObject;
    float shadow = 1.0;
```

```
    vec3 prevColor = primaryObject.color;

    for (int i = 1; i <= bounceVar; i++)
    {
        rayDir                  = reflect(rayDir, normal);
        RayHit t                = reflectedRay(pos + normal * 0.001, rayDir)
            ;
        pos                     = pos + rayDir * t.hitpoint;
        normal                  = GetNormal(pos);

        if (t.hitpoint == -1.0)
            t.color             = vec3(0.36,0.36,0.60) - (rayDir.y * 0.2);
        else
            t.color             = getPointLight(t.color, normal, pos);

        if (t.id == 7 && prevObject.material != MATTE && i < 3)
        {
            vec3 shadowRayOrigin = pos + normal * 0.02;
            vec3 shadowRayDir = light.position - pos;
            shadow = softshadow(shadowRayOrigin, shadowRayDir, 2.0);
            color *= shadow / i;
        }

        if (prevObject.material == MATTE)
            continue;
        else
            color              += t.color * prevColor / i;

        prevColor               = t.color;
        prevObject              = t;
    }

    return color;
}

float softshadow(vec3 ro, vec3 rd, float k)
{
    float res = 1.0;
    float t = 0.0;
    for (int i = 0; i < 16; i++)
    {
        RayHit h = sdf(ro + rd * t);
        if(h.hitpoint < 0.001)
            return 0.05;

        res = min(res, k * h.hitpoint / t);
        t += h.hitpoint;
    }

    return res;
}

vec3 render(vec3 rayOrigin, vec3 rayDir)
{
    vec3 color = vec3(0.30, 0.36, 0.60) - (rayDir.y * 0.2);
    RayHit t = RayMarch(rayOrigin, rayDir);
    float shadow = 1.0;

    if (t.hitpoint != -1.0) {

        vec3 pos          = rayOrigin + rayDir * t.hitpoint;
        vec3 normal       = GetNormal(pos);
```

```
        vec3 prevColor   = t.color;
        color            = t.color;//normal * vec3(0.5) + vec3(0.5);
        color            = getPointLight(color, normal, pos);

        if (t.id == 7) // Everything bellow is applied only for the floor
            plane
        {
            vec3 shadowRayOrigin = pos + normal * 0.02;
            vec3 shadowRayDir    = light.position - pos;
            shadow               = softshadow(shadowRayOrigin, shadowRayDir,
                2.0);
            color               *= shadow;
            color                = pow(color, vec3(0.4545));
            return color;
        }

        if (bounceVar > 0)
            color = bounce(rayDir, pos, normal, color, t);

    }
    // Gamma Correction
    color = pow(color, vec3(0.4545));
    // Visualize depth
    // color = vec3(1.0-t.hitpoint*0.075);
    return color;
}

vec3 getPointLight(vec3 color, vec3 normal, vec3 pos)
{
    vec3 ambient      = light.ambient;
    vec3 viewDir      = normalize(pos - camera.pos.xyz);

    vec3 lightDir     = normalize(light.position - pos);
    float NtoL_vector = max(dot(normal, lightDir), 0.0);
    vec3 diffuse      = light.diffuse * NtoL_vector;

    vec3 reflectDir   = reflect(lightDir, normal);
    float spec        = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);

    vec3 specular     = light.specular * spec;

    float distance    = length(light.position - pos);
    float attenuation = 1.0 / (light.constant + light.linear * distance +
        light.quadratic * (distance * distance));

    diffuse  *= attenuation;
    ambient  *= attenuation;
    specular *= attenuation;

    vec3 final = color * (diffuse + ambient + specular);
    return final;
}

vec3 GetNormal(vec3 pos)
{
    float c = sdf(pos).hitpoint;

    vec2 eps_zero = vec2(0.001, 0.0);
    // clang-format off
    return normalize(vec3(sdf(pos + eps_zero.xyy).hitpoint,
                          sdf(pos + eps_zero.yxy).hitpoint,
                          sdf(pos + eps_zero.yyx).hitpoint) - c);
```

```
    // clang-format on
}


void main()
{
    vec4 pixel;
    vec4 finalColor = vec4(0);
    // Compute global x, y coordinates utilizing local group ID
    const ivec2 tile_xy      = ivec2(gl_WorkGroupID);
    const ivec2 thread_xy    = ivec2(gl_LocalInvocationID);
    const ivec2 pixel_coords = tile_xy * TILE_SIZE + thread_xy;
    // ivec2 pixel_coords = ivec2(gl_GlobalInvocationID.xy);

    ivec2 dims = imageSize(img_output);
    float x    = (float(pixel_coords.x * 2 - dims.x) / dims.x);
    float y    = (float(pixel_coords.y * 2 - dims.y) / dims.y);

    vec2 uv = vec2(x, y);

    // 4xMSAA
    // Anti aliasing on-off
    if (AA)
    {
        uv.x += 0.25 / dims.x;
        uv.y += 0.25 / dims.y;
        Ray r = castRay(uv);
        pixel = vec4(render(r.origin, r.dir), 1.0);
        finalColor += pixel;

        uv.x += 0.75 / dims.x;
        uv.y += 0.25 / dims.y;
        r = castRay(uv);
        pixel = vec4(render(r.origin, r.dir), 1.0);
        finalColor += pixel;

        uv.x += 0.25 / dims.x;
        uv.y += 0.75 / dims.y;
        r = castRay(uv);
        pixel = vec4(render(r.origin, r.dir), 1.0);
        finalColor += pixel;

        uv.x += 0.75 / dims.x;
        uv.y += 0.75 / dims.y;
        r = castRay(uv);
        pixel = vec4(render(r.origin, r.dir), 1.0);
        finalColor += pixel;

        finalColor /= 4;
    }
    else
    {
        Ray r = castRay(uv);
        finalColor = vec4(render(r.origin, r.dir), 1.0);
    }

    imageStore(img_output, pixel_coords, finalColor);
}
```

Βιβλιογραφία

[1] E. Haines N. Hoffman A. Pesce M. Iwanicki S. Hillaire T. Akenine-Möller. ***Real Time Rendering Fourth Edition***. AK Peters Ltd., 2018.

[2] T. Theoharis G. Papaioannou N. Platis N. Patrikalakis. ***Graphics & Visualization*** *Principles & Algorithms*. AK Peters Ltd., 2008.

[3] Greg Humphreys Matt Pharr. ***Physically Based Rendering*** *Second Edition*. Elsevier, Inc., 2010.

[4] Joey de Vries. ***LearnOpenGL***. `https://learnopengl.com/`. Accessed on 29-2-2020. July 2014.

[5] TheCherno. ***Writing a Shader in OpenGL***. `https://www.youtube.com/watch?v=71BLZwRGUJE`. Accessed on 14-5-2019. Oct. 2017.

[6] Anton Gerdelan. ***An Introduction to Compute Shaders***. `https://antongerdelan.net/opengl/compute.html`. Accessed on 29-2-2020. Oct. 2016.

[7] Iñigo Quilez. ***Rendering Worlds with two Triangles***. `https://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf`. Accessed on 29-2-2020. July 2008.

[8] The Art of Code. ***Ray Marching for Dummies!*** `https://www.youtube.com/watch?v=PGtv-dBi2wE`. Accessed on 17-3-2020. Dec. 2018.

[9] Huw Bowles AJ Weeks. ***Raymarching Workshop Course Outline***. `https://github.com/electricsquare/raymarching-workshop`. Accessed on 19-3-2020. Nov. 2019.

[10] Cambridge in Colour. ***Understanding Gamma Correction***. `https://www.cambridgeincolour.com/tutorials/gamma-correction.htm`. Accessed on 20-3-2020. 2005.

[11] StanEpp. ***A camera for a ray tracer***. https://github.com/StanEpp/OpenGL_Raytracing/blob/master/src/Camera.hpp. Accessed on 19-5-2020. July 2019.

[12] Joey de Vries. ***Basic Lighting in OpenGL***. https://learnopengl.com/Lighting/Basic-Lighting. Accessed on 5-6-2020. July 2014.

[13] Scratchapixel. ***Introduction to Shading (Reflection, Refraction and Fresnel)***. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel. Accessed on 15-6-2020.

[14] Iñigo Quilez. ***soft shadows in raymarched SDFs***. https://iquilezles.org/www/articles/rmshadows/rmshadows.htm. Accessed on 16-6-2020. 2010.

[15] slimyfrog. ***GS Rubber Floor with Shadows***. https://www.shadertoy.com/view/3df3DH. Accessed on 20-6-2020. Dec. 2018.

[16] MJP. ***A Quick Overview of MSAA***. https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/. Accessed on 6-7-2020. Oct. 2012.

[17] Iñigo Quilez. ***Distance Functions***. https://iquilezles.org/www/articles/distfunctions/distfunctions.htm. Accessed on 14-7-2020. Mar. 2013.