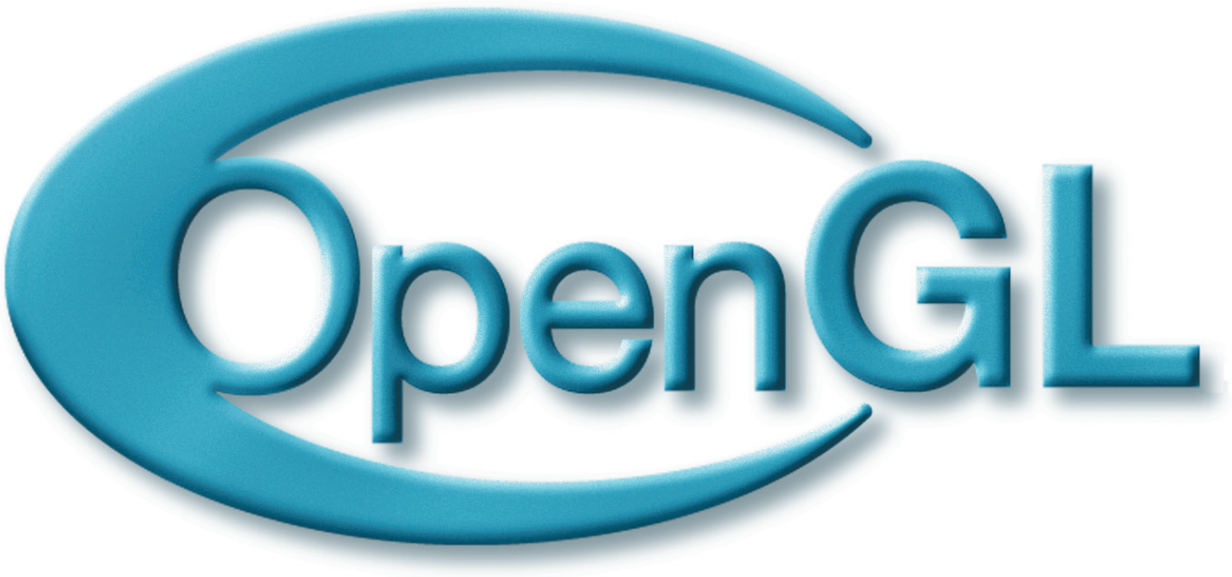


ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΓΡΑΦΙΚΩΝ

3D γραφικά με το OpenGL

ΜΕΡΟΣ 1ο Αυτή η σειρά άρθρων αποσκοπεί στη γνωριμία των αναγνωστών με τον προγραμματισμό 3D γραφικών και συγκεκριμένα με τα realtime 3D γραφικά, χρησιμοποιώντας το OpenGL API...

ΤΟΥ ΓΙΑΝΝΗ ΤΣΙΟΜΠΙΚΑ
Ο Γιάννης είναι προγραμματιστής. Στον ελεύθερο χρόνο του γράφει ελεύθερο λογισμικό και κυρίως επικεντρώνεται στον προγραμματισμό γραφικών. Διαβάστε τη συνέντευξή του στο τεύχος 22.

Όταν μιλάμε για 3D γραφικά, αναφερόμαστε σε διάφορους αλγορίθμους που χρησιμοποιούμε, ώστε από μία μαθηματική περιγραφή ενός τρισδιάστατου περιβάλλοντος να δημιουργήσουμε μία εικόνα που να αναπαριστά αυτό το 3D περιβάλλον στην οθόνη του υπολογιστή. Λέγοντας ότι θα ασχοληθούμε συγκεκριμέναμε realtime γραφικά, εννοούμε ότι μας ενδιαφέρει αυτές οι εικόνες να υπολογίζονται σε κλάσματα του δευτερολέπτου, ώστε να μπορούμε να αλληλεπιδρούμε με τον 3D κόσμο και να βλέπουμε άμεσα αυτές τις αλλαγές στην οθόνη μας. Κλασικό παράδειγμα χρήσης realtime 3D γραφικών είναι τα computer games, όπου τα γραφικά ανανεώνονται συνεχώς, για να δώσουν την αίσθηση τις κίνησης στον παίκτη.

Το πρώτο πρόβλημα που καλούμαστε να αντιμετωπίσουμε, πριν περάσουμε στους αλγορίθμους rendering, είναι το πώς θα αναπαραστήσουμε μία 3D σκηνή στο πρόγραμμάμας. Ο πιο συνηθισμένος τρόπος είναι να χρησιμοποιήσουμε μία σειρά από πολύγωνα, που προσεγγίζουν την επιφάνεια κάθε αντικειμένου (βλ. σχήμα 1). Αυτό το representation μάς παρέχει απέραντη ευελιξία να αναπαραστήσουμε οποιαδήποτε 3Dεπιφάνεια θέλουμε, σε ό,τι βαθμό προσέγγισης θέλουμε.

Τεχνικές

Υπάρχουν δύο βασικές τεχνικές για το rendering 3D γραφικών. Η πιο απλή μέθοδος λέγεται ray-tracing. Το πρόγραμμα "ρίχνει" ακτίνες για κάθε pixel και υπολογίζει τα σημεία τομής αυτών των ακτίνων με τα διάφορα αντικείμενα της σκηνής. Αν και τα αποτελέσματα του ray-tracing μπορούν να είναι πολύ ρεαλιστικά, η διαδικασία είναι αρκετά χρονοβόρα, γι' αυτό δεν χρησιμοποιείται στα realtime γραφικά.

Η δεύτερη τεχνική βασίζεται στο polygon rasterization. Ουσιαστικά, προβάλλουμε τα 3D πολύγωνα των αντικειμένων στο 2D επίπεδο της εικόνας και μετά χρωματίζουμε τα pixels που περιέχονται σε κάθε πολύγωνο με το κατάλληλο χρώμα. Με αυτή την τεχνική θα ασχοληθούμε, γιατί είναι αρκετά γρήγορη, ώστε να χρησιμεύει στα realtime προγράμματα γραφικών και είναι υλοποιημένη κατά μεγάλο μέρος στις εξειδικευμένες κάρτες γραφικών που έχουμε στους υπολογιστές μας τα τελευταία χρόνια.

Τι είναι το OpenGL;

Το OpenGL είναι ένα API για το rendering 3D γραφικών με τη μέθοδο του polygon rasterization. Φτιάχτηκε από τη Silicon Graphics (SGI), στις αρχές του '90, ως μία απόπειρα να δημιουργηθεί ένα standard interface για τις rendering δυνατότητες των διαφόρων graphics workstations. Παρόλο που ξεκίνησε από την SGI, το OpenGL αναπτύχθηκε ως ανεξάρτητο πρότυπο (vendor-independent standard), το οποίο κατευθύνεται από μία επιτροπή αποτελούμενη από διάφορες εταιρείες και οργανισμούς που ασχολούνται με το χώρο, την OpenGL Architecture Review Board (ARB).

Αντίθετα με το προηγούμενο αντίστοιχο API της Silicon Graphics, το IrisGL, το οποίο ήταν άρρηκτα συνδεδεμένο με το X Window System, το OpenGL είναι window-system agnostic. Δεν ασχολείται, δηλαδή, με δημιουργία παραθύρων, event handling (input) κ.λπ., αλλά βασίζεται σε κάποιο ξεχωριστό window system specific glue layer (GLX), για να συνδέσει το OpenGL rendering context με κάποιο παράθυρο.

Παρόλα αυτά, σε αυτά τα άρθρα δεν θα ασχοληθούμε με το GLX

και το X Window System απευθείας, μια και για κάτι τέτοιο θα χρειαζόταν ξεχωριστό άρθρο, αλλά θα χρησιμοποιήσουμε μία απλή και εύχρηστη cross-platform βιβλιοθήκη, ονόματι GLUT (GL Utility Toolkit), που αναλαμβάνει να μας ανοίξει OpenGL παράθυρα και παρέχει ένα απλό callback interface για event handling.

Θα δούμε πρώτα ένα παράδειγμα χρήσης της βιβλιοθήκης GLUT, το οποίο ανοίγει ένα παράθυρο 800x600 και χειρίζεται κάποια events. Επίσης, σε αυτό καλούμε μία συνάρτηση του OpenGL, για να καθαρίσουμε την εικόνα, γεμίζοντάς τη με μαύρο χρώμα. Αυτό θα πρέπει να το κάνουμε κάθε φορά που ανανεώνουμε την εικόνα, αλλιώς θα βλέπουμε "σκουπίδια" από τα προηγούμενα καρέ, οπότε το τοποθετούμε στην αρχή του display callback, το οποίο καλείται αυτόματα από το GLUT κάθε φορά που χρειάζεται να ανανεωθεί η εικόνα στο παράθυρό μας. Επίσης, θα παρατηρήσετε ότι στο τέλος της συνάρτησης display, καλούμε και τη συνάρτηση glutSwapBuffers. Αυτό είναι απαραίτητο για να εμφανιστούν όσα ζωγραφίσαμε στην οθόνη, λόγω του ότι χρησιμοποιούμε double buffering, κάτι που σημαίνει ότι ζωγραφίζουμε σε ένα μη ορατό κομμάτι μνήμης και μόνο αφού τελειώσουμε, το εμφανίζουμε στην οθόνη.

```
#include <stdio.h>
#include <ctype.h>
#include <GL/glut.h> /* glut.h also includes <GL/gl.h>
(the OpenGL header) */
void display(void);
void reshape(int x, int y);
void keyb(unsigned char key, int x, int y);
int main(int argc, char **argv)
{
  /* initialize glut, and create a window */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH |
  GLUT_DOUBLE);
  glutInitWindowSize(800, 600);
  glutCreateWindow("OpenGL window");
  /* register event callback handlers */
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyb);
  /* enter the glut main event handling loop, this never
  returns */
  glutMainLoop();
  return 0;
}
/* the display callback is called when we need to redraw
the graphics in our * OpenGL window (due to X11 Expose
event, or call to
glutPostRedisplay).
*/
void display(void)
{
  /* clear the image (by default clears to black) */
  glClear(GL_COLOR_BUFFER_BIT);

  /* since we requested a double-buffered visual
  (GLUT_DOUBLE flag at
  * glutInit), we need to "show" the back buffer by calling
  glutSwapBuffers.
  */
  glutSwapBuffers();
}
/* the reshape callback is called when our window is
resized */
void reshape(int x, int y)
{
  printf("window resized: %dx%d\n", x, y);
}
/* the keyboard callback is called when a key is pressed
*/
void keyb(unsigned char key, int x, int y)
{
  if(isprint(key)) {
    printf("pressed: '%c'\n", key);
  } else {
    printf("pressed: 0x%x\n", key);
  }
}
```

Για να κάνουμε compile το παραπάνω πρόγραμμα, το οποίο έχουμε, π.χ., σε ένα αρχείο gl1.c, καλούμε τον C compiler του συστήματος ως εξής:

```
cc -o gl1 gl1.c -lGL -lglut
```

Μετασχηματισμοί

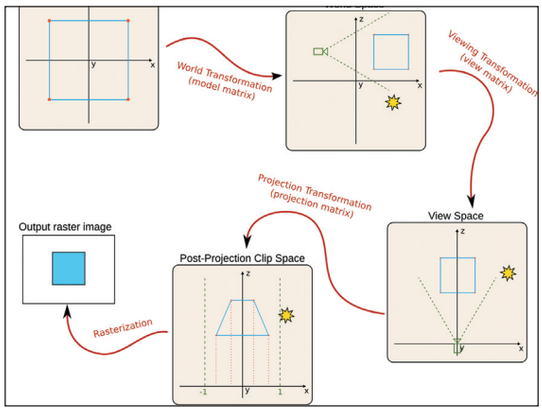
Πριν προχωρήσουμε παραπέρα, πρέπει να πούμε λίγα πράγματα για το πώς τα πολύγωνα των αντικειμένων φτάνουν να ζωγραφιστούν στην οθόνη. Οι κορυφές (vertices) που ορίζουν κάθε πολύγωνο (τα οποία είναι, φυσικά, τρισδιάστατα διανύσματα), περνούν από μία σειρά μετασχηματισμών, μέχρι να καταλήξουν να προβληθούν στο επίπεδο της εικόνας **(βλ. σχήμα 1).**

Κατ' αρχάς, όλα τα αντικείμενα είθισται να ορίζονται σε ένα δικό τους σύστημα συντεταγμένων, το οποίο έχει το κέντρο των αξόνων του στο κέντρο του αντικειμένου ή όπου βολεύει καλύτερα κατά περίπτωση, το οποίο ονομάζεται local coordinate system.

Με κάθε αντικείμενο συσχετίζουμε και έναν μετασχηματισμό που το προσανατολίζει και το τοποθετεί καταλλήλως στο ευρύτερο σύστημα συντεταγμένων της σκηνής, το οποίο λέγεται world space.

Κατόπιν ορίζουμε έναν μετασχηματισμό που φέρνει όλα τα αντικείμενα της σκηνής από το world space στο view space. Αυτό είναι ένα σύστημα συντεταγμένων, στο οποίο η εικονική "camera" βρίσκεται στο κέντρο των αξόνων και "κοιτάει" προς τον z άξονα. Αυτό το κάνουμε, γιατί έτσι πλέον το image plane στο οποίο θέλουμε να προβάλουμε τα πολύγωνά μας, είναι παράλληλο ως προς το xy επίπεδο, κάνοντας την προβολή πολύ πιο απλή μαθηματικά, απ' ό,τι μία προβολή σε ένα τυχαία προσανατολισμένο επίπεδο. Μάλιστα όντας σε view space, μπορούμε να δημιουργήσουμε ισομετρικές προβολές των αντικειμένων, αν απλώς αγνοήσουμε τη συντεταγμένη z.

Στη συνέχεια, οι vertices προβάλλονται στο επίπεδο της εικόνας με μία, συνήθως, προοπτική προβολή, που κάνει τα αντικείμενα να μικραίνουν όσο απομακρύνονται. Αυτό μπορεί να υπολογιστεί με απλή τριγωνομετρία και όμοια τρίγωνα, αλλά για διάφορους λόγους στους οποίους δεν θα αναφερθώ τώρα (γιατί ξεφεύγουν από το πλαίσιο αυτού του άρθρου) χρησιμοποιούμε έναν μετασχηματισμό σε



Σχήμα 1: Η διαδικασία (pipeline) του rendering.

ΓΡΗΓΟΡΗ ΣΥΜΒΟΥΛΗ

Στον τελικό κώδικα του tutorial (gl3.c), δοκιμάστε να περιστρέψετε τον κύβο κατά 25 μοίρες γύρω από τον x άξονα (1 0 0). Δείτε τι συμβαίνει, αν τοποθετήσετε την καινούργια glRotatef κλήση πριν ή μετά από την υπάρχουσα. Η αντιμεταθετική ιδιοτητα δεν ισχύει στον πολλαπλασιασμό πινάκων...

ομογενείς συντεταγμένες, που "αντιστοιχίζει" τις vertices στο λεγόμενο homogenous clip space.

Τελικά, έχοντας προβάλει τις vertices των πολυγώνων στο 2D επίπεδο, θεωρούμε το διάστημα [-1, 1] και στους δύο άξονες ως τα όρια της εικόνας και το κάνουμε map με το viewport transformation, στα διαστήματα [0, width) και [0, height), οριζόντια και κάθετα αντίστοιχα, ώστε να αποκτήσουμε συντεταγμένες πολυγώνων σε pixels, για να γεμίσουμε μετά τα απαραίτητα pixels με το κατάλληλο χρώμα.

Ως γνωστόν, γραμμικοί μετασχηματισμοί σε τρισδιάστατα συστήματα συντεταγμένων μπορούν να αναπαρασταθούν με πίνακες 3x3. Ομως, ένας πολύ χρήσιμος μετασχηματισμός, η παράλληλη μεταφορά κατά διάνυσμα, δεν είναι γραμμικός μετασχηματισμός, αλλά ανήκει στην ευρύτερη κατηγορία των affine transformations.

Γι' αυτό το λόγο (και για το projection που προαναφέραμε), χρησιμοποιούμε ομογενείς συντεταγμένες. Δηλαδή, αντί για 3D διανύσματα, χρησιμοποιούμε 4D διανύσματα με τη συντεταγμένη w=1, καθώς και αντίστοιχα 4x4 πίνακες μετασχηματισμού.

Το OpenGL κρατάει, ως μέρος της κατάστασής του, 4x4 πίνακες για τους παραπάνω μετασχηματισμούς. Η εφαρμογή θέτει κάθε φορά τους κατάλληλους πίνακες στο OpenGL και αυτό αναλαμβάνει να μετασχηματίσει τις vertices που του δίνουμε με αυτούς, κάτι που συχνά αναλαμβάνει το hardware και γίνεται ταχύτατα.

Για να θέσουμε έναν πίνακα στο OpenGL, κατ' αρχάς καλούμε τη συνάρτηση glMatrixMode, για να ορίσουμε ποιον από τους πίνακες θέλουμε να αλλάξουμε και κατόπιν καλούμε τη συνάρτηση glLoadMatrixf, η οποία παίρνει ως παράμετρο τη διεύθυνση ενός array από 16 floats. Οι παράμετροι της glMatrixMode που μας ενδιαφέρουν, είναι οι: GL_MODELVIEW (concatenation των world και view matrices) και GL_PROJECTION. Υπενθύμιση σε όσους δεν θυμούνται τη γραμμική άλγεβρα: ο συνδυασμός δύο μετασχηματισμών (concatenation) γίνεται με πολλαπλασιασμό των δύο πινάκων που τους ορίζουν.

Το viewpoint transformation ορίζεται ξεχωριστά με το glViewport function, το οποίο παίρνει 4 floats με το origin και το μέγεθος σε pixels της εικόνας. Αυτή την συνάρτηση χρειάζεται να την καλέσουμε κάθε φορά που αλλάζει το μέγεθος του παραθύρου στο οποίο ζωγραφίζουμε (όταν το GLUT καλεί το reshape callback).

Για να μη χρειάζεται συνεχώς να κατασκευάζουμε πίνακες μετασχηματισμού και να τους "ταΐζουμε" στο OpenGL με την glLoadMatrixf, υπάρχουν διάφορες βοηθητικές συναρτήσεις που μπορούμε να χρησιμοποιήσουμε. Η glLoadIdentity θέτει έναν μοναδιαίο πίνακα, ενώ οι συναρτήσεις glTranslatef, glRotatef και glScalef δημιουργούν και πολλαπλασιάζουν στο τρέχοντα πίνακα μετασχηματισμού έναν άλλον: translation (μεταφορά), rotation (περιστροφή γύρω από διάνυσμα) ή scaling transformation. Προσοχή: ο πολλαπλασιασμός πινάκων δεν είναι αντιμεταθετικός, οπότε η σειρά με την οποία καλούμε αυτές τις συναρτήσεις, παίζει ρόλο...

Για να ζωγραφίσουμε έναν απλό κύβο, όπως θα κάνουμε εντός ολίγου, δεν είναι απαραίτητο να κάνουμε διαχωρισμό του world από το view matrix. Μπορούμε να αγνοήσουμε εντελώς το world space και να δουλέψουμε σε view space. Ας πούμε ότι θέλουμε να ζωγραφίσουμε τον κύβο μας, περιστραμμένο κατά 20 μοίρες γύρω από τον άξονα y του local coordinate system του και τοποθετημένο 5 μονάδες μακριά στον z άξονα, ώστε να τον βλέπει η camera που, όπως είπαμε, βρίσκεται στο origin και κοιτάει προς το -z. Τότε θα πρέπει να καλέσουμε τις εξής συναρτήσεις στην display(), πριν δώσουμε τα πολύγωνα του κύβου στο OpenGL:

```
glMatrixMode(GL_MODELVIEW); /* modelview matrix */
glLoadIdentity(); /* transformation matrix = identity */
glTranslatef(0, 0, -5); /* μεταφορά 5 μονάδες προς το Z */
glRotatef(20, 0, 1, 0); /* περιστροφή 20 μοιρών στον Y */
```

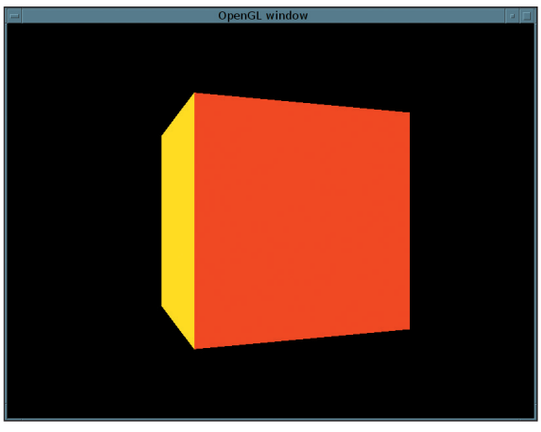
Επίσης, πρέπει να θέσουμε και ένα κατάλληλο projection matrix. Τα καλά νέα είναι ότι δεν χρειάζεται να το υπολογίσουμε και να το θέσουμε με την glLoadMatrixf, αφού υπάρχει μία βολική συνάρτηση στη βοηθητική βιβλιοθήκη GLU, που πάντα έρχεται με το OpenGL, η οποία κατασκευάζει τον πίνακα και τον πολλαπλασιάζει στον τρέχοντα. Η πρώτη παράμετρος της gluPerspective είναι το κάθετο πεδίο θέασης σε μοίρες, η δεύτερη είναι ο λόγος πλάτους προς μήκους της εικόνας (ώστε να υπολογίσει το οριζόντιο πεδίο θέασης), ενώ οι δύο τελευταίες παράμετροι ορίζουν τα near και far clipping planes. Οτιδήποτε είναι πιο κοντά από το near και πιο μακριά από το far clipping plane σε view space, κόβονται. Αυτό είναι απαραίτητο κατ' αρχάς για να μη ζωγραφιστούν αντικείμενα που είναι πίσω από το viewpoint, αλλά και για τη λειτουργία του z-buffering, για το οποίο θα μιλήσουμε σε λίγο. Μια και το projection matrix μας, όπως είπαμε, εξαρτάται από το μέγεθος του παραθύρου στο οποίο ζωγραφίζουμε, μία καλή θέση γι' αυτόν τον κώδικα είναι το reshape callback. Αυτή η συνάρτηση καλείται εγγυημένα από το GLUT και στην αρχή του προγράμματός μας, μόλις δημιουργηθεί το παράθυρο, οπότε θα κληθεί οπωσδήποτε πριν ζωγραφίσουμε για πρώτη φορά. Βάλτε τον κώδικα στη reshape():

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45.0, (float)x / (float)y, 1.0, 1000.0);
```

Είμαστε έτοιμοι, λοιπόν, να δώσουμε τα πολύγωνα του κύβου στο OpenGL. Αυτό θα το κάνουμε με μία σειρά από glVertex3f κλήσεις, που ορίζουν τα διανύσματα θέσης των κορυφών των πολυγώνων του κύβου στο local coordinate system. Αυτές τις κλήσεις θα τις κάνουμε μέσα σε ένα glBegin/glEnd μπλοκ, περνώντας στην glBegin την παράμετρο GL_QUADS, για να καθορίσουμε ότι οι vertices που δίνουμε, θέλουμε να ομαδοποιηθούν ανά 4 σε τετράπλευρα πολύγωνα (quadrilaterals). Μαζί με τα διανύσματα θέσης, οι vertices στο OpenGL κουβαλάνε και άλλες ιδιότητες, όπως ένα χρώμα για κάθε vertex. Αυτές τις πληροφορίες τις δίνουμε στο OpenGL με παρόμοια calls (π.χ., glColor3f), και ισχύουν για όλες τις vertices από το σημείο που δόθηκαν και μετά, μέχρι να αλλαχτούν. Ετσι, στον κύβο μας δίνουμε επίσης από ένα χρώμα σε κάθε πλευρά, καλώντας την glColor3f με τις ίδιες 3 παραμέτρους red, green και blue για κάθε vertex αυτής της πλευράς.

Προσθέτουμε, λοιπόν, τον παρακάτω κώδικα στην display(), μετά τα glRotate/glTranslate calls που ορίζουν το μετασχηματισμό του κύβου μας και πριν από την glutSwapBuffers που προβάλλει το αποτέλεσμα:

```
glBegin(GL_QUADS);
/* πίσω πλευρά (-Z) */
glColor3f(0, 0, 1);
glVertex3f(1, -1, -1);
glVertex3f(-1, -1, -1);
glVertex3f(-1, 1, -1);
glVertex3f(1, 1, -1);
/* πάνω πλευρά (+Y) */
glColor3f(0, 1, 1);
glVertex3f(-1, 1, 1);
glVertex3f(1, 1, 1);
```



Εικόνα 1: Ενας 3D κύβος - το "Hello World" του OpenGL.

```
glVertex3f(1, 1, -1);
glVertex3f(-1, 1, -1);
/* κάτω πλευρά (-Y) */
glColor3f(1, 0, 1);
glVertex3f(-1, -1, -1);
glVertex3f(1, -1, -1);
glVertex3f(1, -1, 1);
glVertex3f(-1, -1, 1);
/* δεξιά πλευρά (+X) */
glColor3f(0, 1, 0);
glVertex3f(1, -1, 1);
glVertex3f(1, -1, -1);
glVertex3f(1, 1, -1);
glVertex3f(1, 1, 1);
/* αριστερή πλευρά (-X) */
glColor3f(1, 1, 0);
glVertex3f(-1, -1, -1);
glVertex3f(-1, -1, 1);
glVertex3f(-1, 1, 1);
glVertex3f(-1, 1, -1);
/* εμπρός πλευρά (+Z) */
glColor3f(1, 0, 0);
glVertex3f(-1, -1, 1);
glVertex3f(1, -1, 1);
glVertex3f(1, 1, 1);
glVertex3f(-1, 1, 1);
glEnd();
```

Δοκιμάστε να τρέξετε τον κώδικα και θα πρέπει να δείτε την κόκκινη πλευρά του κύβου και λίγο από την κίτρινη, λόγω της περιστροφής 20 μοιρών γύρω από τον άξονα y, όπως ορίσαμε **(βλ. εικόνα 1).** Για να κάνετε compile το παραπάνω, θα χρειαστείτε επίσης το διακόπτη -lGLU, μια και πρέπει να γίνει link η βοηθητική βιβλιοθήκη GLU για την gluPerspective() που χρησιμοποιήσαμε.

Περιστροφή κατά βούληση!

Ας δούμε τώρα τι πρέπει να κάνουμε, για να μπορούμε να περιστρέφουμε τον κύβο interactively με το πληκτρολόγιο. Θα χρησιμοποιήσουμε τα πλήκτρα <, > για να περιστρέψουμε τον κύβο μας δεξιόστροφα ή αριστερόστροφα κατά βούληση. Για να το πετύχουμε αυτό, πρέπει να φτιάξουμε μία global μεταβλητή που θα κρατάει την παρούσα γωνία του κύβου, την οποία θα αυξομειώνουμε με το πάτημα αυτών των δύο πλήκτρων, και θα χρησιμοποιήσουμε αυτή τη μεταβλητή στο glRotatef call, αντί για τη σταθερή γωνία των 20 μοιρών. Οπότε, προσθέστε την εξής μεταβλητή στην αρχή του προγράμματός μας, πριν από τη συνάρτηση main:

```
float angle;
```

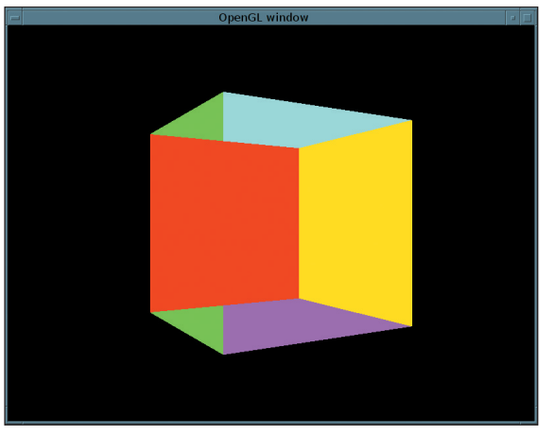
Αλλάξτε το glRotatef call σε:

```
glRotatef(angle, 0, 1, 0);
```

Και αλλάξτε τη συνάρτηση keyb ως εξής:

```
void keyb(unsigned char key, int x, int y)
{
  switch(key) {
  case ',':
    angle -= 2;
    glutPostRedisplay();
    break;
  case '.':
    angle += 2;
    glutPostRedisplay();
    break;
  }
}
```

Οι κλήσεις στην glutPostRedisplay λένε στο GLUT ότι θέλουμε να ζωγραφιστεί ξανά η εικόνα, ώστε να δούμε τις αλλαγές. Το GLUT με τη σειρά του καλεί την display συνάρτησή μας, όπου θέτουμε το σωστό rotation και ζωγραφίζουμε ξανά τον κύβο.



Εικόνα 2: Περιστρέφοντας τον κύβο, το πρόβλημα είναι προφανές...

Αν δοκιμάσετε το πρόγραμμα σε αυτή τη φάση, θα διαπιστώσετε ότι κάτι πάει στραβά, όταν περιστρέφετε τον κύβο, ώστε οι πίσω πλευρές να έρθουν μπροστά **(βλ. Εικόνα 2)**. Το πρόβλημα οφείλεται στο ότι το OpenGL ακολουθεί πιστά τις οδηγίες μας και ζωγραφίζει τα πολύγωνα ακριβώς με τη σειρά που του λέμε. Έτσι, το κόκκινο πολύγωνο, που μετά την περιστροφή είναι πίσω από το μπλε, ζωγραφίζεται τελευταίο όπως ορίσαμε στο glBegin/glEnd block και έτσι γράφεται πάνω από το μπλε.

Υπάρχουν διάφορες λύσεις σε αυτό το πρόβλημα, το οποίο ονομάζεται visible surface determination, η καθεμία με τα πλεονεκτήματα και τα μειονεκτήματά της. Για παράδειγμα, θα μπορούσαμε να αποθηκεύσουμε τα πολύγωνά μας σε ένα array, να το κάνουμε sort κατά βάθος σε κάθε frame και να ζωγραφίσουμε τα πολύγωνα back-to-front. Αυτό είναι γνωστό ως το painter's algorithm, γιατί μοιάζει με τον τρόπο που ο ζωγράφος ζωγραφίζει τα αντικείμενα στον καμβά και, αν και δουλεύει, στην περίπτωσή μας, σπάει σε πολύπλοκες σκηνές, με αντικείμενα που τέμνουν το ένα το άλλο.

Η πιο απλή και καθολική λύση στο visible surface determination problem, την οποία θα χρησιμοποιήσουμε, λέγεται z-buffering και δουλεύει ως εξής: για κάθε pixel της εικόνας, κρατάμε μία τιμή σε ένα ξεχωριστό buffer (zbuffer ή depth buffer), στο οποίο γράφουμε το βάθος του pixel στην εικόνα (το z coordinate σε view space). Μετά, κατά το γέμισμα των πολυγώνων, υπολογίζουμε σε κάθε pixel του πολυγώνου το βάθος του (με linear interpolation των z των τριών vertices) και το συγκρίνουμε με το z που έχει ήδη γραφτεί στον zbuffer για αυτό το pixel. Αν το z του συγκεκριμένου pixel του πολυγώνου είναι μικρότερο από αυτό του zbuffer, μόνο τότε γράφουμε το pixel αυτό στην εικόνα. Αν είναι μεγαλύτερο, σημαίνει ότι έχει ήδη γραφτεί κάτι που είναι πιο κοντά στο θεατή προηγουμένως, οπότε δεν το κάνουμε overwrite. Φυσικά, πριν αρχίσουμε να ζωγραφίζουμε οτιδήποτε, πρέπει να κάνουμε initialize τον depth buffer στην τιμή που αντιστοιχεί στο μακρύτερο δυνατό σημείο.

Ολα τα παραπάνω, φυσικά, τα αναλαμβάνει το OpenGL. Εμείς αρκεί να ενεργοποιήσουμε το GL_DEPTH_TEST και να καθαρίσουμε τον depth buffer στην αρχή κάθε frame.

Οπότε, προσθέστε κάπου, π.χ., στην main, πριν από το glutMainLoop call, το εξής:

```
glEnable(GL_DEPTH_TEST);
```

Και αλλάξτε το glClear call στην αρχή της display συνάρτησης σε:

```
glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
```

Αν τρέξετε ξανά το πρόγραμμα, θα πρέπει ο κύβος να φαίνεται σωστά από κάθε γωνία. Ο τελικός κώδικας είναι στο αρχείο gl3.c... ■

ΕΠΙΛΟΓΟΣ

Εδώ τελειώνει αυτό το πρώτο μέρος της σειράς OpenGL tutorials. Στο επόμενο τεύχος θα μιλήσουμε για το matrix stack, το οποίο θα μας βοηθήσει να ξεχωρίσουμε το world από το view transformation του modelview matrix και να τοποθετήσουμε περισσότερα αντικείμενα στην σκηνή, καθώς και για το φωτισμό των αντικειμένων από φωτεινές πηγές που μπορούμε να ορίσουμε για μεγαλύτερο ρεαλισμό.