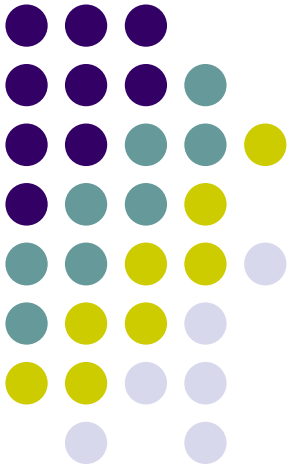
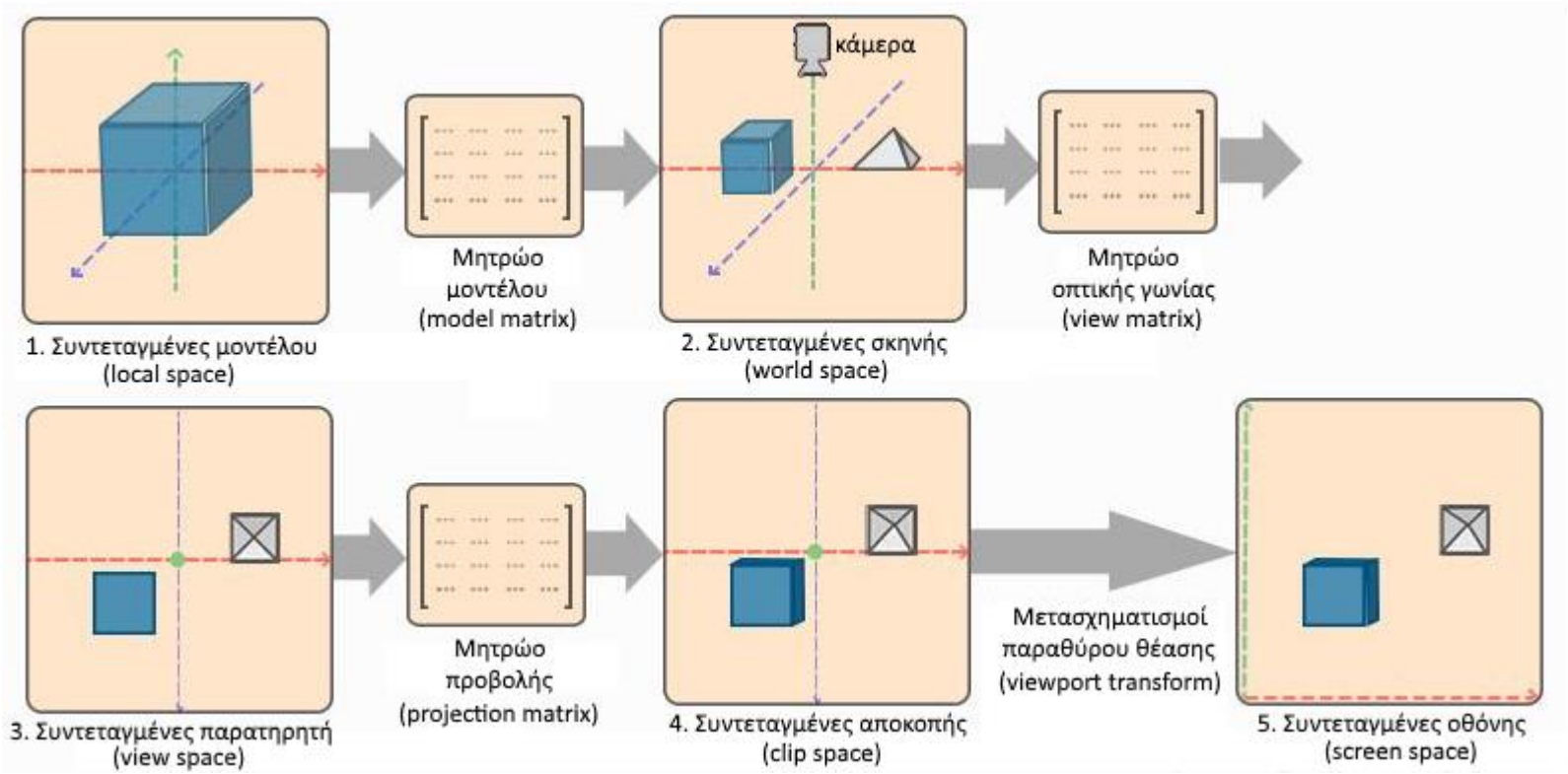
Ανακαλύπτοντας τις 3Δ δυνατότητες της OpenGL

Ματθές Δημήτριος



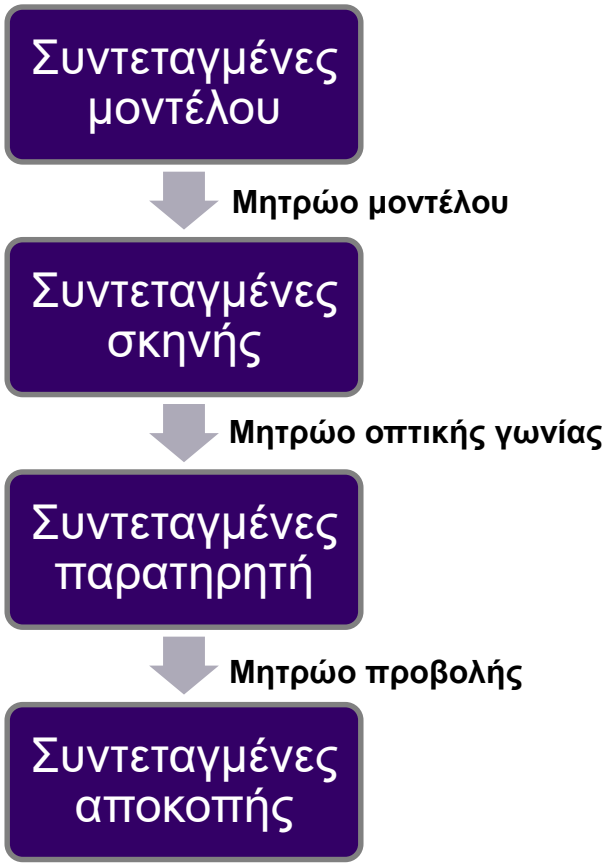
Σύστημα συντεταγμένων και μετασχηματισμοί

- Η OpenGL χρησιμοποιεί διαφορετικά συστήματα συντεταγμένων και για να αποδώσει τον 3Δ κόσμο και τα αντικείμενά του στην οθόνη κάνει ενδιάμεσους μετασχηματισμούς. Τα συστήματα και οι μετασχηματισμοί αυτοί είναι:



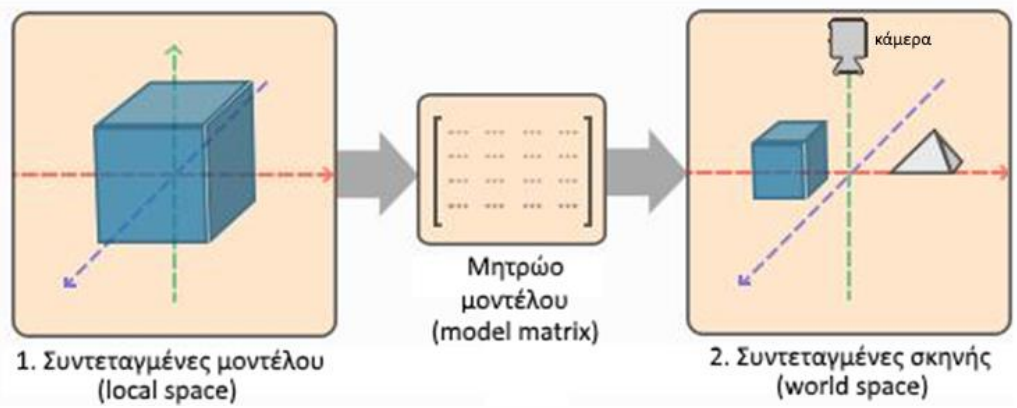
Σύστημα συντεταγμένων και μετασχηματισμοί

- Ο λόγος που χρησιμοποιούμε ενδιάμεσα συστήματα συντεταγμένων είναι ότι κάποιες λειτουργίες γίνονται ευκολότερα σε κάποια από αυτά παρά σε ένα ενιαίο.
- Οι μετασχηματισμοί γίνονται με τη χρήση τριών βασικών μητρώων (matrices, πίνακες):
  - Μοντέλου (Model)
  - Οπτικής γωνίας (View)
  - Προβολής (Projection)



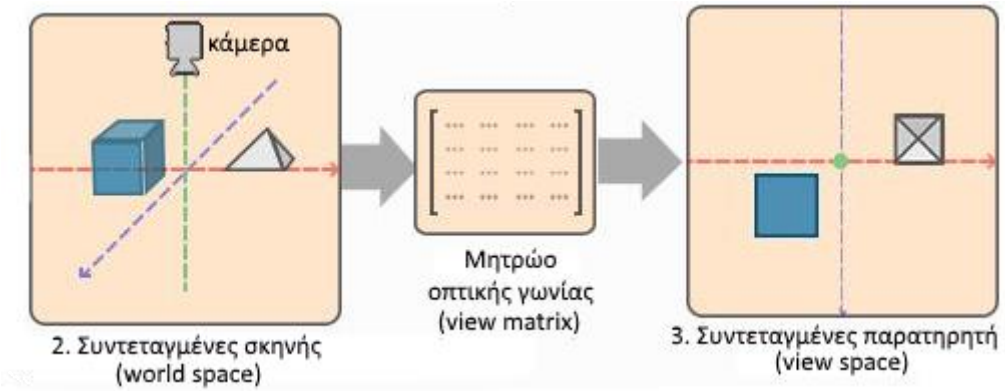
Συντεταγμένες μοντέλου → Συντεταγμένες σκηνής

- Οι συντεταγμένες μοντέλου είναι οι συντεταγμένες βάσει των οποίων σχεδιάζουμε το αντικείμενό μας.
- Ξεκινώντας συνήθως από το σημείο (0, 0, 0) καθορίζουμε τα χαρακτηριστικά του αντικειμένου μας, όπως το πλάτος, το ύψος, το βάθος κλπ.
- Δουλεύουμε, προς διευκόλυνσή μας, με τις συντεταγμένες μοντέλου στην αρχή προκειμένου στη συνέχεια να τοποθετήσουμε το αντικείμενό μας στη σκηνή.
- Η τοποθέτηση του αντικειμένου από τις συντεταγμένες μοντέλου στις συντεταγμένες σκηνής γίνεται με τη χρήση του μητρώου μοντέλου συνδυάζοντας ενέργειες όπως η μετατόπιση, η κλιμάκωση, η περιστροφή.



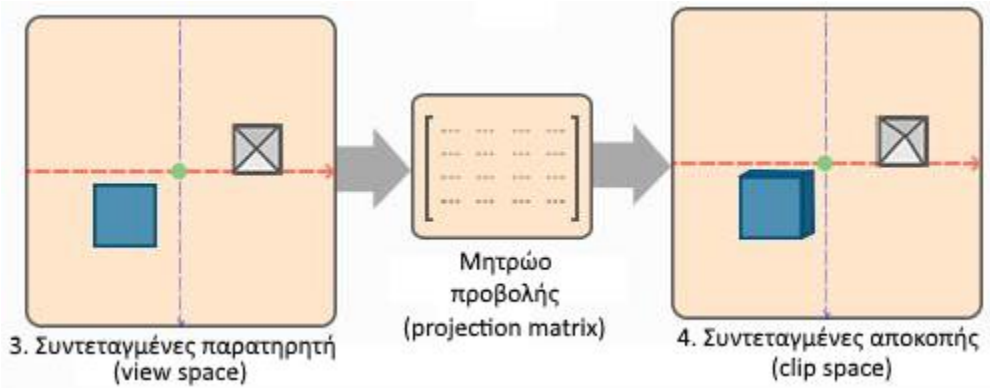
Συντεταγμένες σκηνής → Συντεταγμένες παρατηρητή

- Η σκηνή περιέχει διάφορα αντικείμενα διατεταγμένα στον χώρο.
- Ένας παρατηρητής βλέπει τη σκηνή και τα αντικείμενα αυτά ανάλογα με τη θέση στην οποία βρίσκεται.
- Θα μπορούσαμε να παρομοιάσουμε τον παρατηρητή με μία κάμερα.
- Αν τοποθετήσουμε την κάμερα μπροστά στην σκηνή έχουμε μία οπτική γωνία της σκηνής η οποία είναι διαφορετική από την οπτική γωνία που προκύπτει αν τοποθετήσουμε την κάμερα σε διαφορετική θέση, π.χ. επάνω από την σκηνή.
- Με τη βοήθεια του μητρώου οπτικής γωνίας μετατρέπουμε τις συντεταγμένες σκηνής σε συντεταγμένες παρατηρητή.



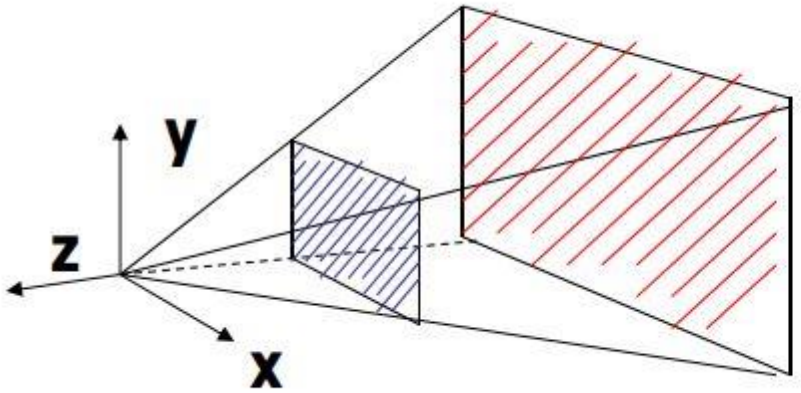
Συντεταγμένες παρατηρητή → Συντεταγμένες αποκοπής

- Κάποια αντικείμενα από αυτά που υπάρχουν μπροστά από τον παρατηρητή (ή αλλιώς την κάμερα) θα πρέπει να μείνουν εκτός της τελικής εικόνας που θα προκύψει.
- Για παράδειγμα, αν κάποιος κοιτάζει ένα δάσος μέσα από ένα παράθυρο βλέπει μόνο ότι του επιτρέπουν τα όρια του παραθύρου κι όχι ολόκληρο το δάσος.
- Ακόμα κι αν δεν υπήρχε το παράθυρο, θα μπορούσε να δει μόνο όσα του επέτρεπε το εύρος των ματιών του.
- Το μητρώο προβολής αποκόπτει την «περιττή» πληροφορία και κρατάει μόνο τον 3Δ χώρο στον οποίο μπορεί να δει ο παρατηρητής.
- Ο 3Δ χώρος χρησιμοποιεί τις λεγόμενες συντεταγμένες αποκοπής. Οτιδήποτε είναι εκτός των συντεταγμένων αυτών απλώς απορρίπτεται.
- Βάσει του χώρου αυτού γίνεται η προβολή (μετατροπή) από τις 3Δ στις 2Δ στην οθόνη.

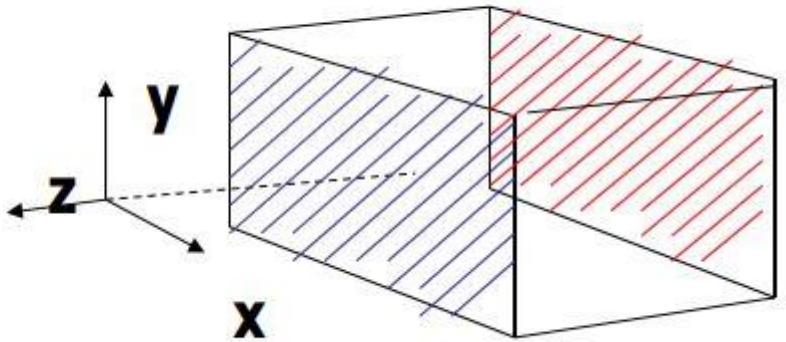


Τύποι προβολών

- Το μητρώο προβολής κάνει χρήση δύο διαφορετικού τύπου 3Δ χώρων για να αποκόψει τα αντικείμενα που βλέπει ο παρατηρητής σε αυτόν.
- Ο πρώτος τύπος είναι ένα κανονικό «κουτί», ένα απλό 3Δ ορθογώνιο παραλληλόγραμμο.
- Ο δεύτερος τύπος είναι μία κόλουρη πυραμίδα (frustum), μία πυραμίδα δηλαδή με «κομμένη μύτη».
- Απόρροια των παραπάνω είναι οι ακόλουθες προβολές:
  - **Ορθογραφική προβολή** (orthographic projection)
  - **Προοπτική προβολή** (perspective projection)
- Κάθε μία από τις προβολές αυτές έχει μειονεκτήματα και πλεονεκτήματα, χρησιμοποιούμε εκείνη που μας βολεύει, ανάλογα με το τι θέλουμε να πετύχουμε.



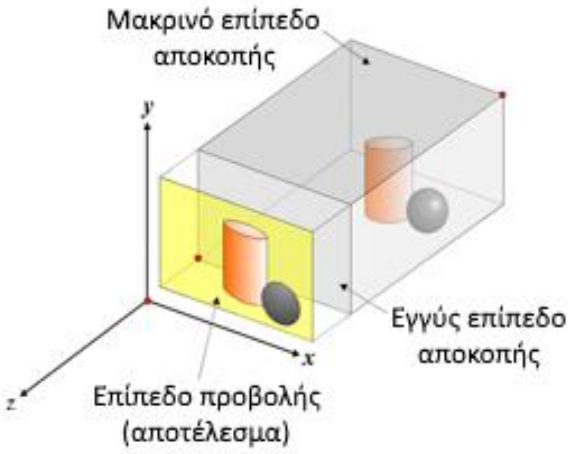
Προοπτική προβολή

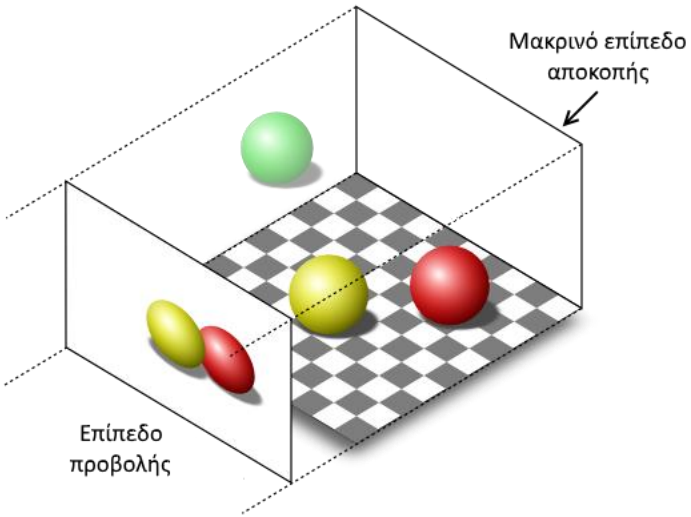


Ορθογραφική προβολή

Ορθογραφική προβολή

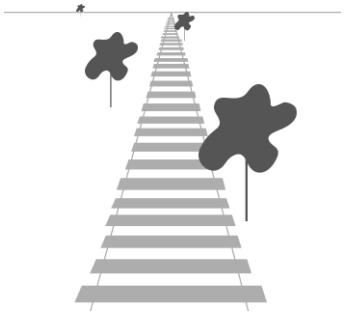
- Μπορούμε να φανταστούμε την **ορθογραφική προβολή** σαν την αποτύπωση του περιεχομένου ενός 3Δ ορθογώνιου παραλληλογράμμου στις 2Δ.
- Αν υποθέσουμε ότι «σαρώνουμε» τον 3Δ αυτό χώρο από το εγγύς επίπεδο αποκοπής προς το μακρινό τότε κάθε σημείο που συναντάμε «αποτυπώνεται» στο επίπεδο προβολής με αναλογία 1-1.
- Άρα αν έχουμε δύο αντικείμενα του ίδιου μεγέθους σε διαφορετικά επίπεδα βάθους τότε αυτά αποτυπώνονται με το ίδιο μέγεθος στο επίπεδο προβολής.
- Το ανθρώπινο μάτι όμως δεν λειτουργεί έτσι. Γνωρίζουμε ότι δύο αντικείμενα με το ίδιο μέγεθος αντικατοπτρίζονται διαφορετικά αν βρίσκονται σε διαφορετικά βάθη: το μακρινό αντικείμενο φαίνεται μικρότερο σε σχέση με το κοντινό.
- Η προβολή αυτή δεν είναι άκρως ρεαλιστική αλλά είναι πολύ χρήσιμη σε προγράμματα όπως ο CAD σχεδιασμός.

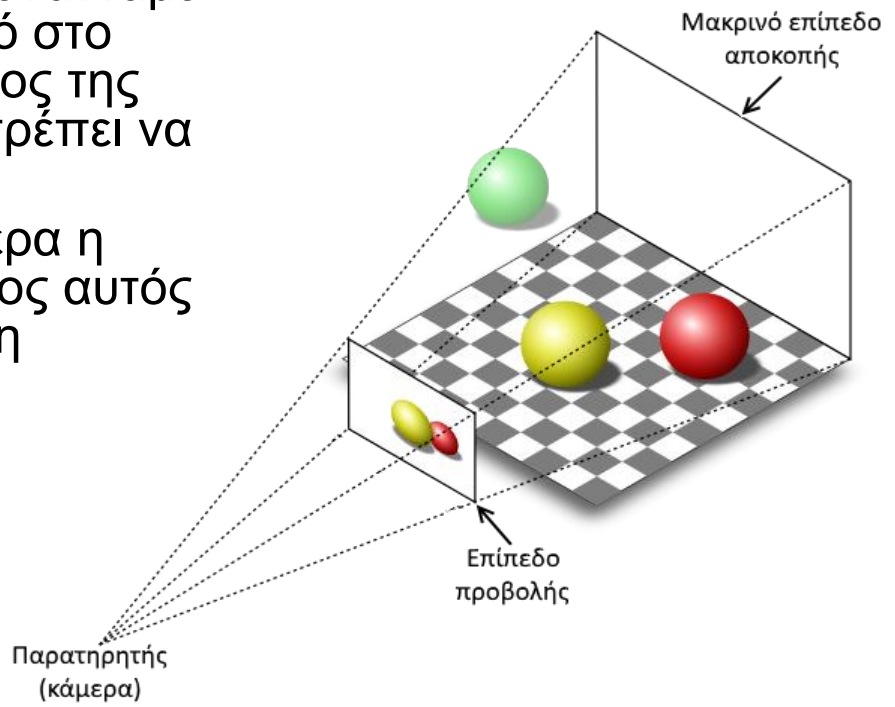




Προοπτική προβολή

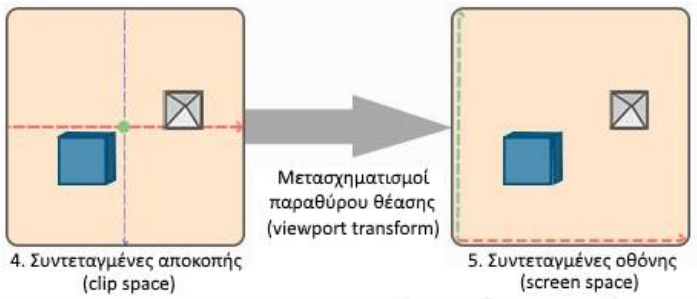
- Για πιο αληθοφανή προβολή χρησιμοποιούμε την **προοπτική προβολή**.
- Ο 3Δ χώρος θυμίζει μία κόλουρη πυραμίδα (frustum).
- Αν «σαρώσουμε» τον χώρο αυτό από το εγγύς επίπεδο αποκοπής προς το αντίστοιχο μακρινό τότε κάθε σημείο που σταδιακά θα συναντάμε θα απεικονίζεται ολοένα και πιο μικρό στο επίπεδο προβολής λόγω του σχήματος της πυραμίδας και των αναλογιών που πρέπει να διατηρηθούν.
- Κατά συνέπεια, επιτυγχάνεται καλύτερα η ψευδαίσθηση του βάθους και ο τρόπος αυτός προβολής συνάδει περισσότερο με τη λειτουργία του ανθρώπινου ματιού.

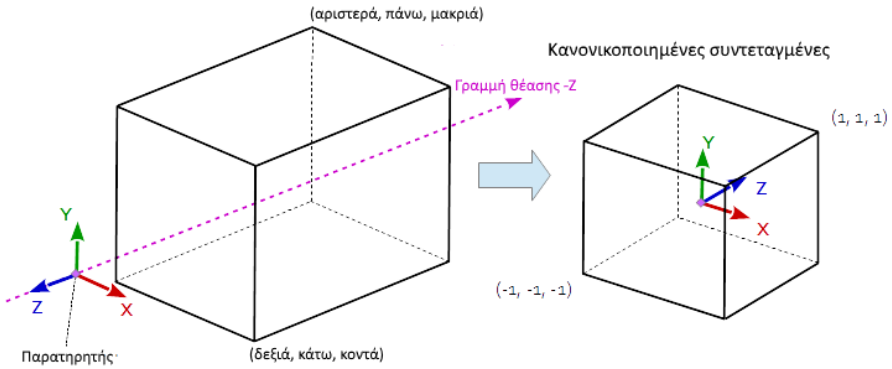


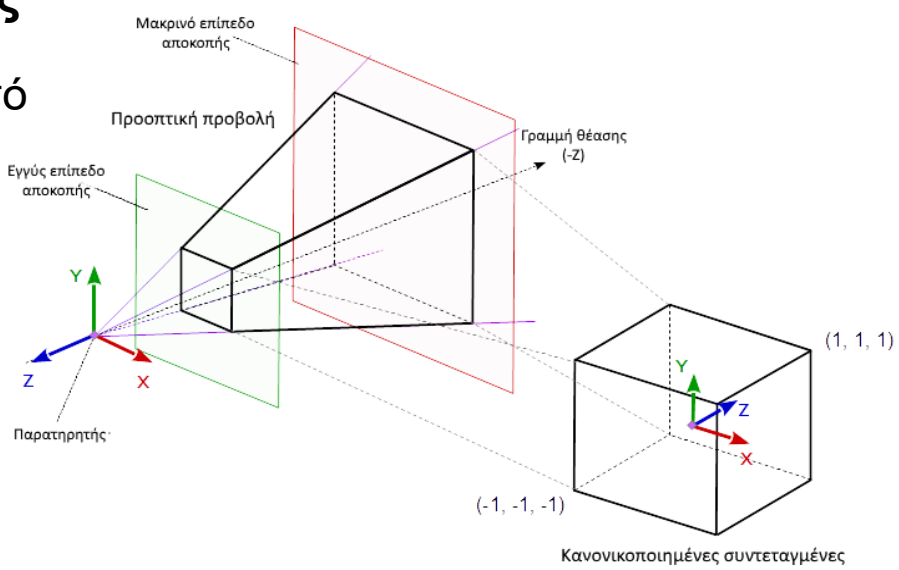


Συντεταγμένες αποκοπής → Συντεταγμένες οθόνης

- Το τελευταίο σύστημα συντεταγμένων είναι η συντεταγμένες οθόνης.
- Ο μετασχηματισμός από τις συντεταγμένες αποκοπής στις συντεταγμένες οθόνης γίνεται αυτόματα από την OpenGL αφού **κανονικοποιήσει** πρώτα τις συντεταγμένες αποκοπής.
- Με τον όρο **κανονικοποίηση** εννοούμε την αντιστοίχιση των συντεταγμένων αποκοπής σε συντεταγμένες με **εύρος από -1.0 έως 1.0**.
- Η κανονικοποίηση γίνεται ανεξάρτητα από τον τύπο της προβολής (ορθογραφική ή προοπτική).







Εναλλαγή μητρώων μοντέλου /οπτικής γωνίας και προβολής

- Όταν πρόκειται να δουλέψουμε με τα μητρώα μοντέλου και οπτικής γωνίας, χρησιμοποιούμε την εντολή:

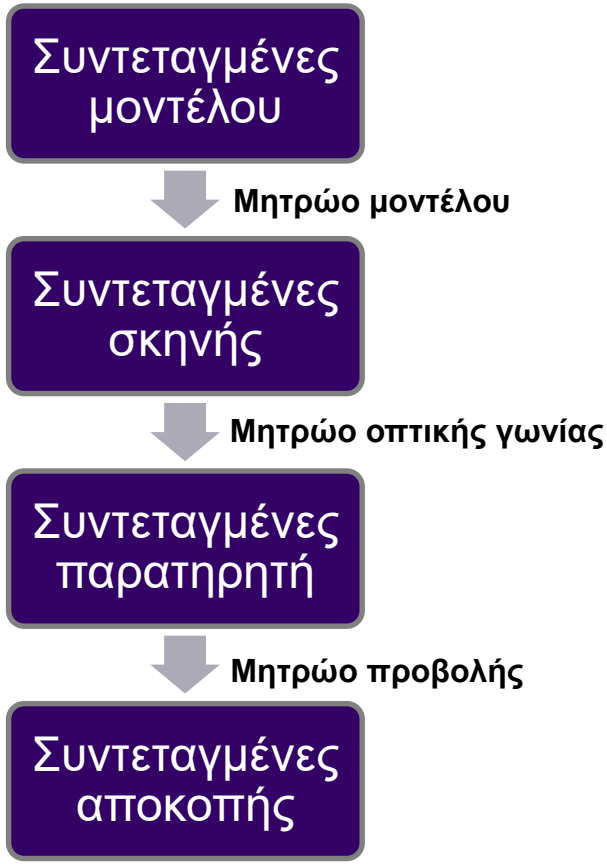
```
glMatrixMode(GL_MODELVIEW);
```

- Οι μετασχηματισμοί που ακολουθούν μετά την εντολή αυτή σχετίζονται με τη δημιουργία αντικειμένων ή σχημάτων και τις ενέργειες που μπορούμε να κάνουμε σε αυτά (μετακίνηση, κλιμάκωση, περιστροφή κλπ).

- Όταν πρόκειται να δουλέψουμε με το μητρώο προβολής, χρησιμοποιούμε την εντολή:

```
glMatrixMode(GL_PROJECTION);
```

- Συνήθως, μετά την εντολή αυτή ακολουθεί η δήλωση του τύπου προβολής (ορθογραφική ή προοπτική) και οι παράμετροί τους.

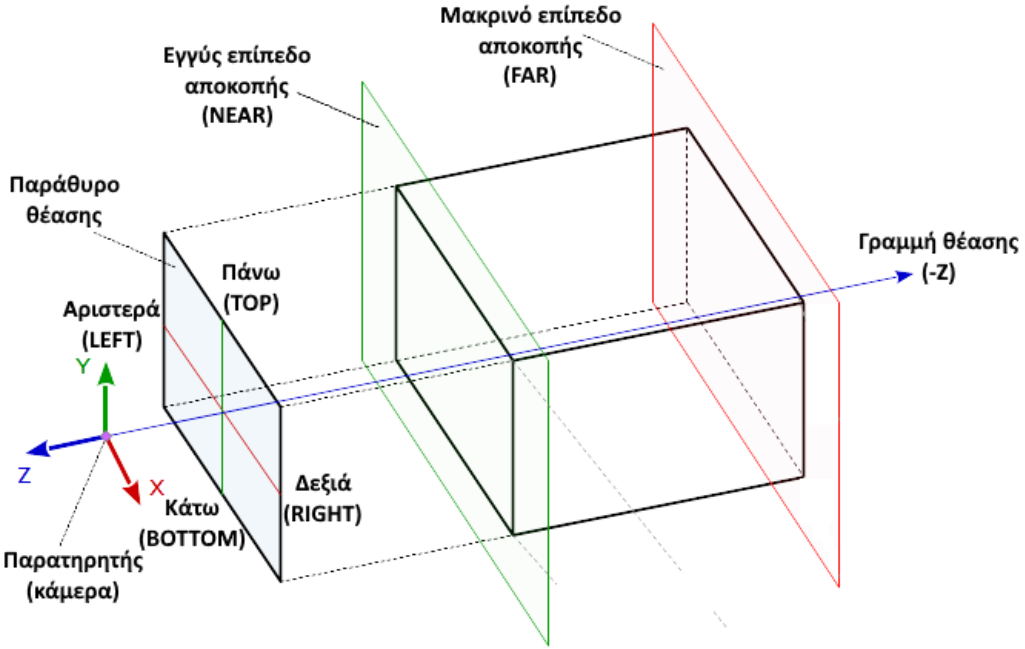


Δήλωση ορθογραφικής προβολής

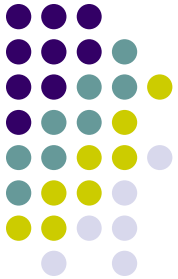
- Η δήλωση ορθογραφικής προβολής γίνεται με την εντολή:

```
glOrtho(LEFT, RIGHT, BOTTOM, TOP, NEAR, FAR);
```

  - **LEFT, RIGHT**: Το αριστερό και το δεξί όριο του ορθογραφικού όγκου θέασης
  - **BOTTOM, TOP**: Το κάτω και το πάνω όριο του ορθογραφικού όγκου θέασης
  - **NEAR, FAR:** Το εγγύς και το μακρινό επίπεδο του ορθογραφικού όγκου θέασης



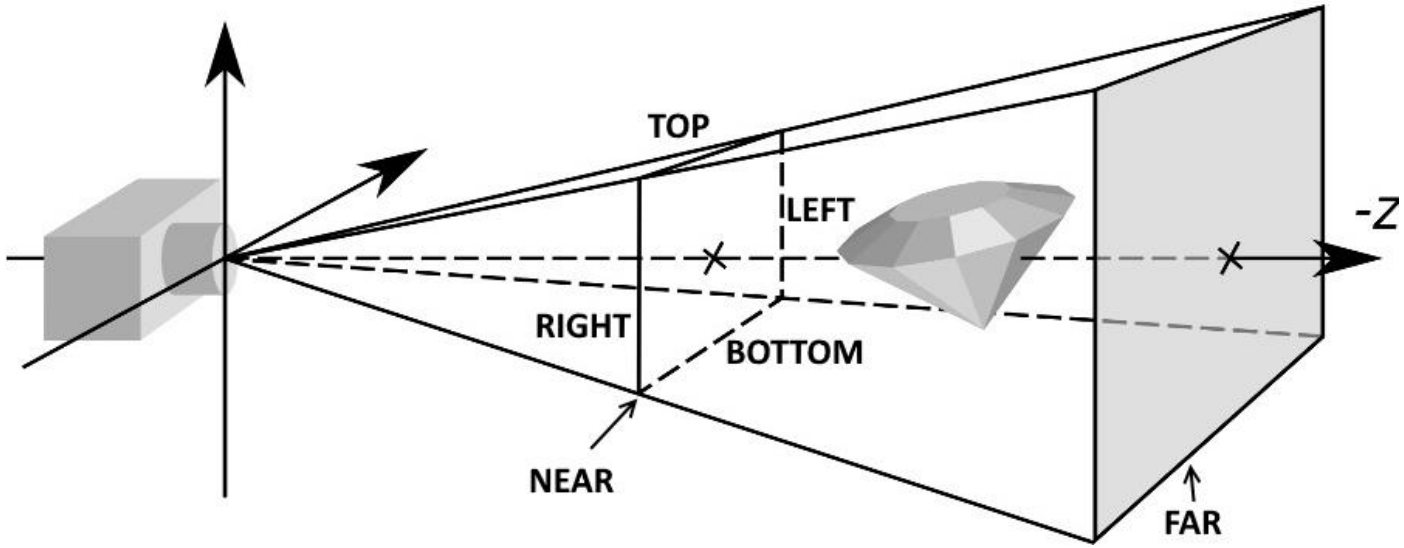
Δήλωση προοπτικής προβολής



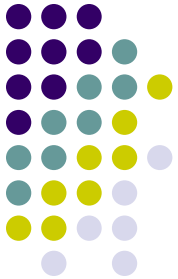
- Η δήλωση μιας προοπτικής προβολής μπορεί να γίνει με 2 εντολές. Η πρώτη είναι η:

```
glFrustum(LEFT, RIGHT, BOTTOM, TOP, NEAR, FAR);
```

  - **LEFT, RIGHT**: Το αριστερό και το δεξί όριο του όγκου θέασης
  - **BOTTOM, TOP**: Το κάτω και το πάνω όριο του όγκου θέασης
  - **NEAR, FAR:** Το εγγύς και το μακρινό επίπεδο του όγκου θέασης. Και οι δύο τιμές πρέπει να είναι θετικοί αριθμοί.



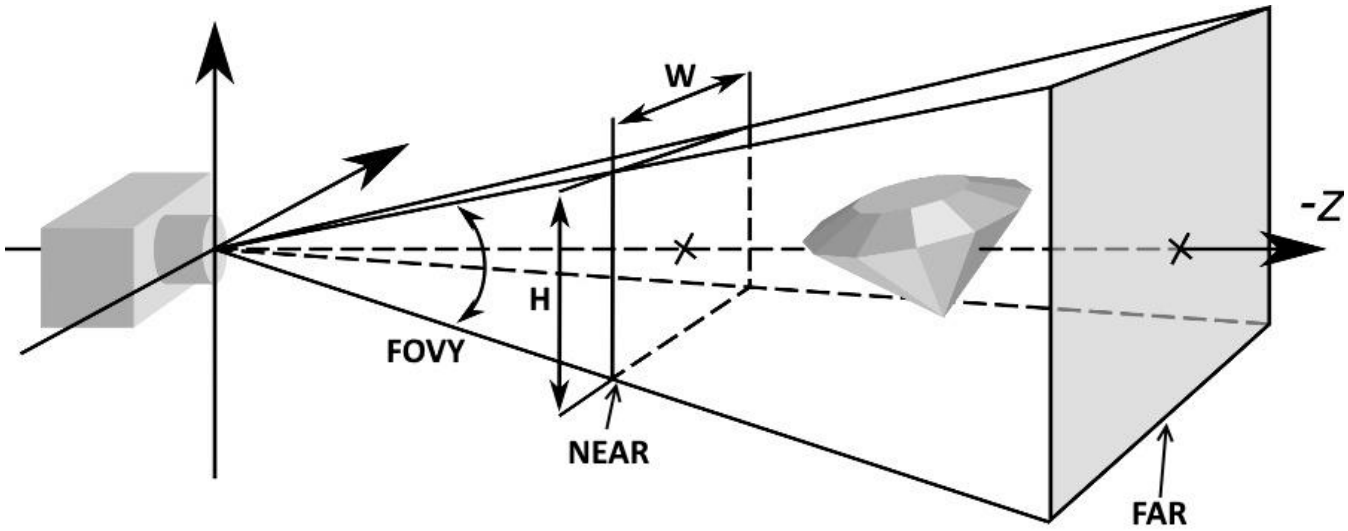
Δήλωση προοπτικής προβολής



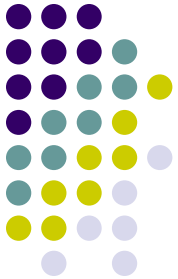
- Η δεύτερη εντολή για τη δήλωση μιας προοπτικής προβολής είναι:

```
gluPerspective(FOVY, ASPECT, NEAR, FAR);
```

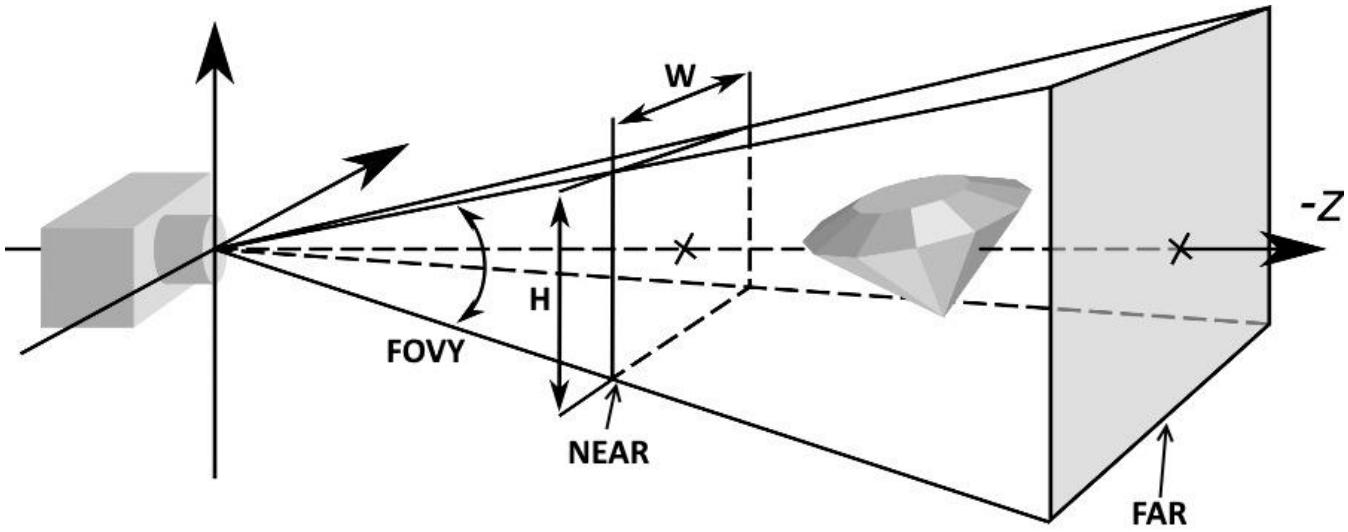
  - **FOVY**: Η γωνία θέασης σε μοίρες κατά τον άξονα Y.
  - **ASPECT**: Καθορίζει την αναλογία των διαστάσεων που του οπτικού πεδίου στην κατεύθυνση X. Η αναλογία αυτή είναι ο λόγος X (πλάτος) προς Y (ύψος).
  - **NEAR, FAR:** Το εγγύς και το μακρινό επίπεδο του όγκου θέασης. Και οι δύο τιμές πρέπει να είναι θετικοί αριθμοί.



Δήλωση προοπτικής προβολής



- Και οι δύο εντολές επιτυγχάνουν το ίδιο αποτέλεσμα.
- Η **gluPerspective()** είναι ελαφρώς πιο εύχρηστη από την **glFrustum()** και γι'αυτό θα την προτιμήσουμε στην κατασκευή του 3Δ κόσμου μας.



Δημιουργώντας τον αρχικό κώδικα σε OpenGL

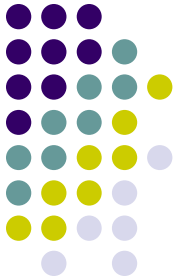
- Για να δημιουργήσουμε ένα παράθυρο στην **OpenGL**, θα πρέπει να γράψουμε τον ακόλουθο κώδικα:

```
#include <GL/glut.h>

void renderGraphics(void)
{

}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(640, 480);
  glutCreateWindow("OpenGL 3D");
  glutDisplayFunc(renderGraphics);
  glutMainLoop();
  return 0;
}
```

Το αποτέλεσμα είναι ένα κενό παράθυρο διαστάσεων 640x480, το οποίο τοποθετείται στις συντεταγμένες {100, 100} επί της οθόνης μας. Έχει τίτλο "**OpenGL 3D**". Το χρωματικό μοντέλο που θα χρησιμοποιηθεί είναι το RGBA (**GLUT_RGBA**) και η σχεδίαση γραφικών σε αυτό θα γίνεται με χρήση διπλού ενταμιευτή* (**GLUT_DOUBLE**). Η συνάρτηση για την ανανέωση των γραφικών είναι η **renderGraphics()**.

διπλός ενταμιευτής: Αγγλ. *double buffer*

Ανανέωση κατά την αλλαγή διαστάσεων του παραθύρου

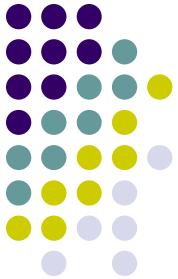


- Θα μπορούσαμε να δηλώσουμε τον τύπο της προβολής που θα χρησιμοποιήσουμε, δηλαδή την προοπτική προβολή **gluPerspective()**, στη συνάρτηση **main()**

- Θεωρείται όμως καλή πρακτική η δήλωσή της να γίνεται στη συνάρτηση η οποία καλείται όταν αλλάζουν οι διαστάσεις του παραθύρου.
- Αυτό γίνεται μέσω της εντολής **glutReshapeFunc()** η οποία λειτουργεί παρόμοια με την **glutDisplayFunc()**.
- Η δηλωθείσα συνάρτηση στην **glutReshapeFunc()** καλείται μία φορά με την εκκίνηση του προγράμματος και εκτελείται κάθε φορά που οι διαστάσεις του παραθύρου μεταβάλλονται.
- Οι παράμετροι που στέλνονται σε αυτή από την **FreeGLUT** είναι το πλάτος και το ύψος του παραθύρου.

```
#include <GL/glut.h>

float fovy  = 45.0f;

void reshapeWindow(int w, int h)
{
  if(h==0) h=1;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glViewport(0, 0, w, h);
  gluPerspective(fovy, (GLfloat)w/(GLfloat)h, 0.1, 100);
  glMatrixMode(GL_MODELVIEW);
}

void renderGraphics(void)
{

}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(640, 480);
  glutCreateWindow("OpenGL 3D");
  glutDisplayFunc(renderGraphics);
  glutReshapeFunc(reshapeWindow);
  glutMainLoop();
  return 0;
}
```

Ανανέωση κατά την αλλαγή διαστάσεων του παραθύρου



- Η εντολή **glLoadIdentity()** φορτώνει τον μοναδιαίο πίνακα.
- Με την κλήση της επαναφέρουμε τα μητρώα των μετασχηματισμών στην αρχική τους κατάσταση.

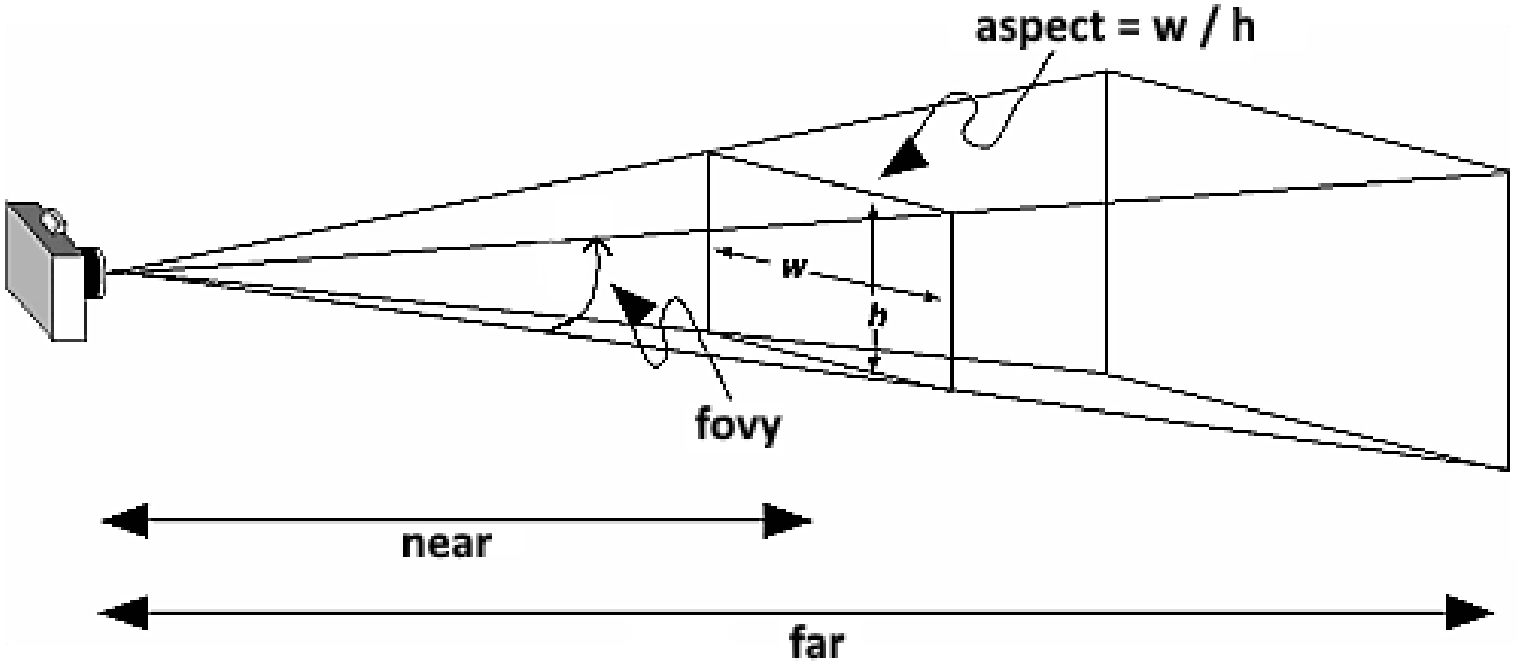
- Η εντολή **glViewport()** χρησιμοποιείται για να θέσουμε τις διαστάσεις του παραθύρου θέασης ίδιες με αυτές του παραθύρου του προγράμματος μας. Αυτό βοηθάει στο να μην αλλοιώνονται οι αναλογίες της εικόνας μας.
- Η εντολή **if(h==0) h=1;** δεν αφήνει το ύψος του παραθύρου να μηδενιστεί και ενδεχομένως να δημιουργήσει προβλήματα στην προβολή.
- Η δήλωση της προοπτικής προβολής γίνεται στο μητρώου προβολής ενώ μετά επανερχόμαστε στο μητρώο μοντέλου/οπτικής γωνίας.

```
#include <GL/glut.h>

float fovy  = 45.0f;

void reshapeWindow(int w, int h)
{
  if(h==0) h=1;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glViewport(0, 0, w, h);
  gluPerspective(fovy, (GLfloat)w/(GLfloat)h, 0.1, 100);
  glMatrixMode(GL_MODELVIEW);
}

void renderGraphics(void)
{

}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(640, 480);
  glutCreateWindow("OpenGL 3D");
  glutDisplayFunc(renderGraphics);
  glutReshapeFunc(reshapeWindow);
  glutMainLoop();
  return 0;
}
```

Ανανέωση κατά την αλλαγή διαστάσεων του παραθύρου

- Η **gluPespective()** στην περίπτωση μας αναλύεται ως εξής:

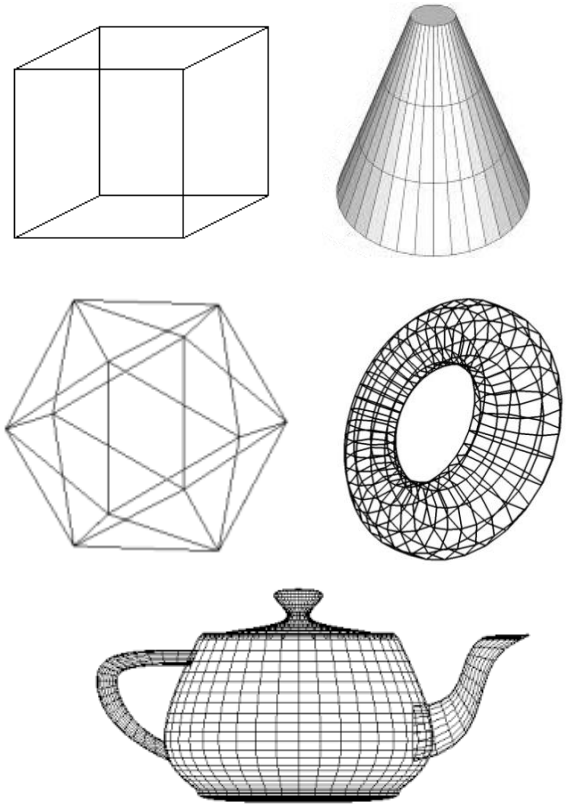
```
                          fovy  aspect   near  far
gluPerspective(45, 640/480, 0.1, 100);
```



Έτοιμα 3Δ μοντέλα

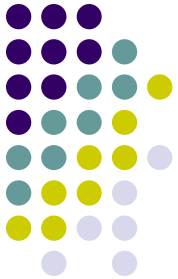
- Η βιβλιοθήκη **FreeGLUT** μας δίνει τη δυνατότητα να χρησιμοποιήσουμε έτοιμα 3Δ μοντέλα χωρίς να χρειαστεί να τα σχεδιάσουμε εξ αρχής.
- Τα μοντέλα αυτά κατατάσσονται σε δύο κατηγορίες:
  - **Μοντέλα πλέγματος** (Wireframe models)
  - **Στερεά μοντέλα** (Solid models)

Μοντέλο	Μοντέλο πλέγματος	Στερεό μοντέλο
Κύβος	glutWireCube()	glutSolidCube()
Κώνος	glutWireCone()	glutSolidCone()
Σφαίρα	glutWireSphere()	glutSolidSphere()
Τόρος	glutWireTorus()	glutSolidTorus()
Τετράεδρο	glutWireTetrahedron()	glutSolidTetrahedron()
Οκτάεδρο	glutWireOctahedron()	glutSolidOctahedron()
Δωδεκάεδρο	glutWireDodecahedron()	glutSolidDodecahedron()
Εικοσάεδρο	glutWireIcosahedron()	glutSolidIcosahedron()
Τσαγιέρα*	glutWireTeapot()	glutSolidTeapot()



** Η τσαγιέρα (Utah Teapot) στα 3Δ γραφικά θεωρείται κάτι αντίστοιχο με το μήνυμα "Hello World" που εμφανίζει κανείς πρώτη φορά στην οθόνη όταν μαθαίνει μία γλώσσα προγραμματισμού.*

Εισάγοντας μία τσαγιέρα στη σκηνή



- Ας δοκιμάσουμε να εισάγουμε στη σκηνή μας ένα 3Δ μοντέλο, π.χ. μία τσαγιέρα. Στη συνάρτηση **renderGraphics()** θα πρέπει:
  1. Να καθαρίσουμε πρώτα την οθόνη με κάποιο χρώμα (έστω λευκό) για να εξαφανιστούν τα όποια γραφικά προϋπήρχαν (εντολές **glClearColor()** και **glClear(GL_COLOR_BUFFER_BIT)**.
  2. Να φορτώσουμε τον μοναδιαίο πίνακα προκειμένου να επαναφέρουμε τον σχεδιασμό μας σε αρχική κατάσταση (συντεταγμένες μοντέλου) χρησιμοποιώντας ως σημείο αναφοράς το {0, 0, 0}.
  3. Να ορίσουμε το χρώμα της «πένας» με την οποία θα σχεδιάσουμε την τσαγιέρα μας (εντολή **glColor3ub()**), έστω μπλε.
  4. Να μετατοπίσουμε το σημείο σχεδιασμού λίγο πιο πίσω (π.χ. -5 μονάδες στο βάθος) προκειμένου το αντικείμενό μας να φαίνεται καλύτερα.
  5. Να εισάγουμε το μοντέλο «τσαγιέρα» με σε μέγεθος 1.0.
  6. Να δώσουμε εντολή στον διπλό ενταμιευτή να φέρει στο προσκήνιο την οθόνη με τα γραφικά που σχεδίασε στο παρασκήνιο με την εντολή **glutSwapBuffers()**.

```
void renderGraphics(void)
{
  glClearColor(1, 1, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  glColor3ub(0, 0, 255);
  glTranslatef(0, 0, -5);
  glutWireTeapot(1.0);
  glutSwapBuffers();
}
```

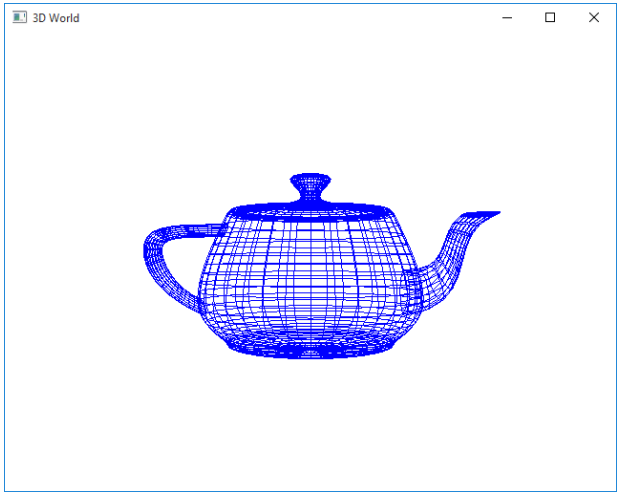
Επαναφορά όλων των μητρώων μετασχηματισμού

Μετατόπιση κατά 5 μονάδες στο βάθος (άξονας Z)

Εισαγωγή μοντέλου «τσαγιέρα» με μέγεθος 1.0

Εισάγοντας μία τσαγιέρα στη σκηνή

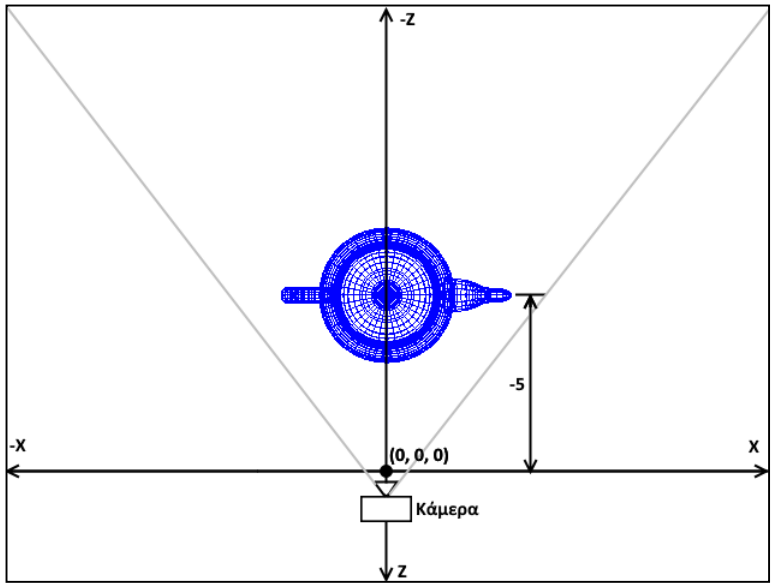
- Αν παρατηρήσουμε τον κώδικα θα διαπιστώσουμε ότι πρώτα πραγματοποιήσαμε τη μετατόπιση και μετά σχεδιάσαμε την τσαγιέρα.
- Είναι πολύ σημαντικό να καταλάβουμε τη σειρά εκτέλεσης των μετασχηματισμών η οποία μπορεί να ερμηνευτεί με δύο τρόπους:
  - Σύμφωνα με τον πρώτο, οι μετασχηματισμοί εκτελούνται με τη σειρά που καλούνται στον κώδικα και επιδρούν στο σύστημα αναφοράς (θεώρηση μεταβαλλόμενου συστήματος αναφοράς). Δηλαδή μετατοπίζεται το σύστημα αναφοράς κι όχι ο παρατηρητής. Κάθε αντικείμενο από εκεί και πέρα τοποθετείται στο νέο σύστημα αναφοράς
  - Σύμφωνα με τον δεύτερο, οι μετασχηματισμοί εκτελούνται με την αντίστροφη σειρά και επιδρούν στα αντικείμενα που έπονται, χωρίς να μεταβάλλεται το σύστημα αναφοράς (θεώρηση αναλλοίωτου συστήματος αναφοράς).
- Οι δύο θεωρήσεις-ερμηνείες είναι ισοδύναμες και μπορείτε να υιοθετήσετε όποια σας βολεύει, αρκεί να ξέρετε τι κάνετε.



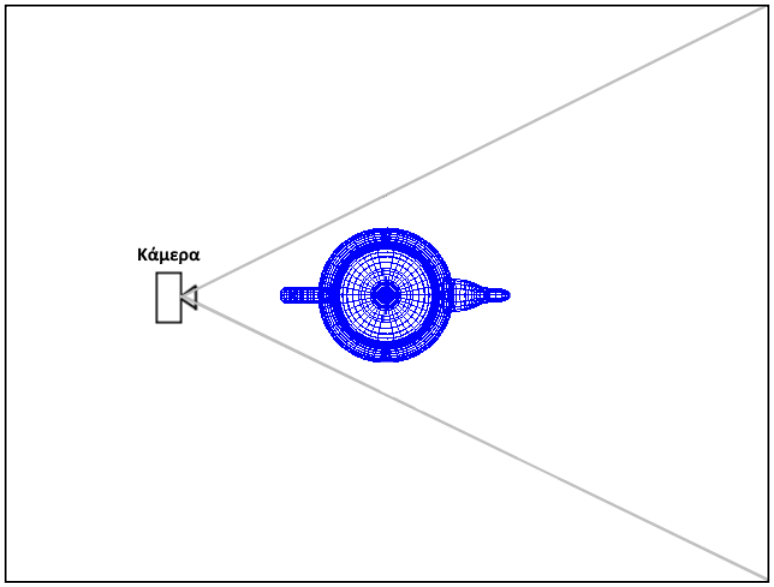

```
void renderGraphics(void)
{
  glClearColor(1, 1, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  glColor3ub(0, 0, 255);
  glTranslatef(0, 0, -5);
  glutWireTeapot(1.0);
  glutSwapBuffers();
}
```

Κοιτώντας την τσαγιέρα από το πλάι

- Αν μπορούσαμε να δούμε τη σκηνή από ψηλά και αν η κάμερα του παρατηρητή ήταν ορατή θα βλέπαμε κάτι σαν το Σχήμα 1:



Σχήμα 1

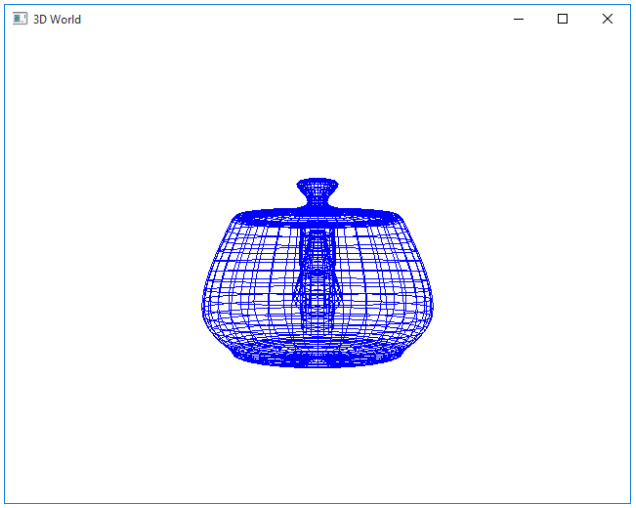


Σχήμα 2

- Πως θα μπορούσαμε να κοιτάξουμε την τσαγιέρα από τα αριστερά (από το χερούλι) και σε απόσταση 5 μονάδων από αυτή με το που ξεκινάει το πρόγραμμά μας (βλ. Σχ. 2); Πως θα μπορούσαμε να το πετύχουμε αυτό προγραμματιστικά;

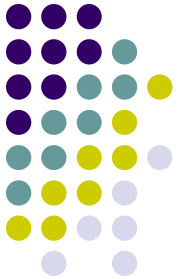
Κοιτώντας την τσαγιέρα από το πλάι

- Για να δούμε την τσαγιέρα από το πλάι αρκεί να εφαρμόσουμε δύο μετασχηματισμούς μετά την αρχικοποίηση των μητρώων:
  - Μία μετατόπιση 5 μονάδων στο βάθος
  - Μία περιστροφή 90^{o} αντίθετα με τους δείκτες του ρολογιού και γύρω από τον άξονα Υ.
- Ας δούμε σταδιακά τι συμβαίνει βάσει της θεώρηση του μεταβαλλόμενου συστήματος αναφοράς (οι άξονες μεταβάλλονται, όχι η κάμερα).



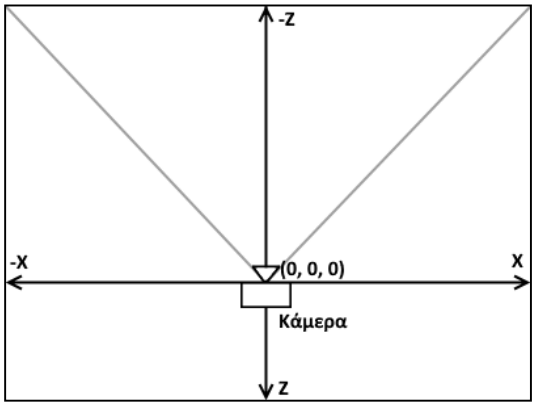

```
void renderGraphics(void)
{
  glClearColor(1, 1, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  glColor3ub(0, 0, 255);
  glTranslatef(0, 0, -5);
  glRotatef(90, 0, 1, 0);
  glutWireTeapot(1.0);
  glutSwapBuffers();
}
```

Κοιτώντας την τσαγιέρα από το πλάι

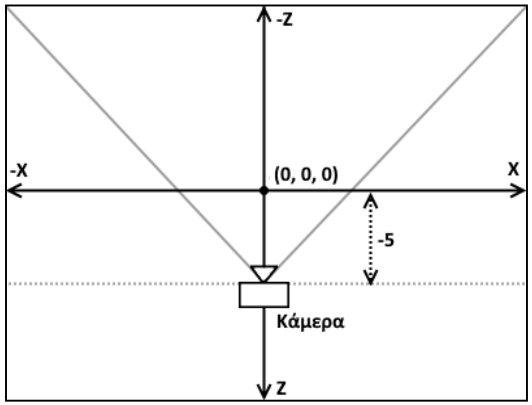


- Με την έναρξη του προγράμματος και με την εκτέλεση της **glLoadIdentity()** η κάμερα είναι σε σταθερή θέση, επί της αρχής των αξόνων (Σχ. 1).
- Με την εντολή **glTranslatef(0, 0, -5)** μετατοπίζεται το σύστημα των αξόνων κατά 5 μονάδες στο βάθος (Σχ. 2).
- Με την εντολή **glRotatef(90, 0, 1, 0)** περιστρέφεται το σύστημα των αξόνων κατά 90^{o} αντίθετα με τους δείκτες του ρολογιού γύρω από τον άξονα Y (Σχ. 3).

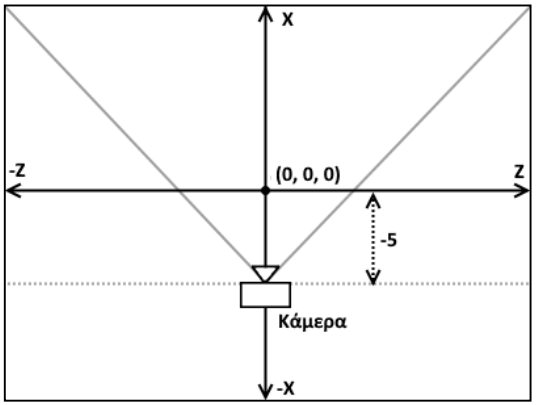
```
void renderGraphics(void)
{
  glClearColor(1, 1, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  glColor3ub(0, 0, 255);
  glTranslatef(0, 0, -5);
  glRotatef(90, 0, 1, 0);
  glutWireTeapot(1.0);
  glutSwapBuffers();
}
```



Σχήμα 1
glLoadIdentity();

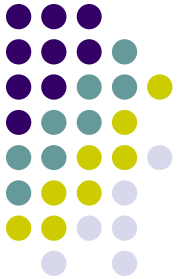


Σχήμα 2
glTranslatef(0, 0, -5);



Σχήμα 3
glRotatef(90, 0, 1, 0);

Περιστρέφοντας την τσαγιέρα



- Με τη λογική που περιγράψαμε και με τις εντολές που χρησιμοποιήσαμε, θα μπορούσαμε να δημιουργήσουμε κίνηση με την τσαγιέρα να περιστρέφεται γύρω από τον άξονά της.
- Το μόνο που πρέπει να κάνουμε είναι σχεδιάζουμε διαρκώς τη σκηνή (δημιουργία συνεχόμενων καρέ) αυξάνοντας κάθε φορά τη γωνία περιστροφής.

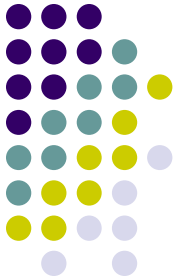
- Για να γίνουν αυτά, στο τμήμα δηλώσεων καθολικών μεταβλητών προσθέτουμε την μεταβλητή **angle**:

```
#include <GL/glut.h>

float fovy  = 45.0f, angle = 0;
```

- Δηλώνουμε στη συνάρτηση **main()** έναν **χρονιστή** οποίος θα καλέσει μια συνάρτηση με όνομα **timer()** με το που θα ξεκινήσει το πρόγραμμά μας.

```
int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(640, 480);
  glutCreateWindow("OpenGL 3D");
  glutDisplayFunc(renderGraphics);
  glutReshapeFunc(reshapeWindow);
  glutTimerFunc(1000.0 / 60.0, timer, 0);
  glutMainLoop();
  return 0;
}
```

Περιστρέφοντας την τσαγιέρα



- Προσθέτουμε τη συνάρτηση **timer()** η οποία θα καλεί την **renderGraphics()** 60 φορές το δευτερόλεπτο.

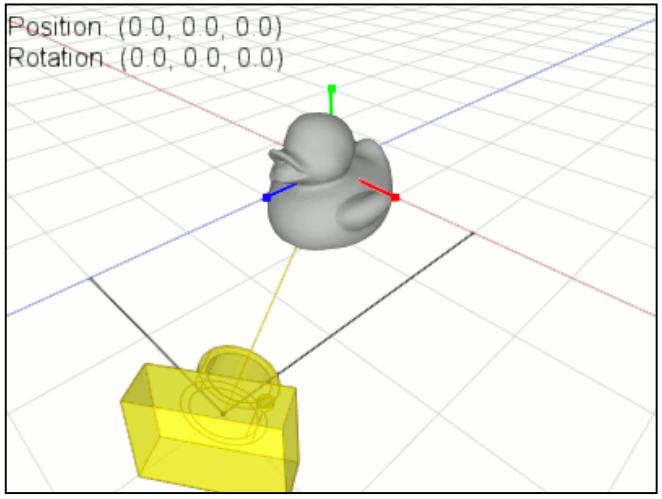
```
void timer(int)
{
    glutPostRedisplay();
    glutTimerFunc(1000.0 / 60.0, timer, 0);
}
```

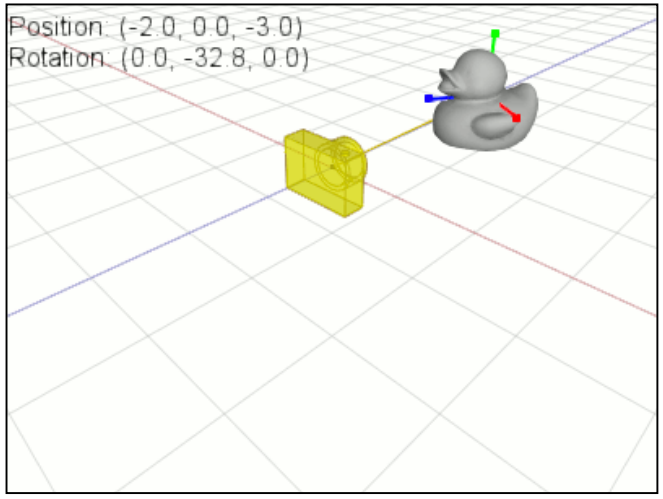
- Τέλος, εντός της **renderGraphics()** αυξάνουμε τη γωνία κατά 1ο κάθε φορά. Όταν ξεπεράσουμε τις 360ο μηδενίζουμε τη μεταβλητή μας.

```
void renderGraphics(void)
{
  glClearColor(1, 1, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  glColor3ub(0, 0, 255);
  glTranslatef(0, 0, -5);
  glRotatef(angle, 0, 1, 0);
  glutWireTeapot(1.0);
  angle += 1;
  if(angle>359) angle=0;
  glutSwapBuffers();
}
```

Κάμερα

- Η OpenGL δεν ενσωματώνει αυτούσια τη λογική της κάμερας.
- Όταν θέλουμε να δούμε τα αντικείμενα που υπάρχουν σε μία 3Δ σκηνή τότε θα πρέπει να δημιουργήσουμε εμείς την οπτική γωνία που θέλουμε μέσω διαδοχικών μετατοπίσεων, περιστροφών και κλιμακώσεων.
- Γενικά, μπορούμε να εξομοιώσουμε την κίνηση μίας κάμερας κινώντας τα αντικείμενα της σκηνής προς την αντίθετη κατεύθυνση δίνοντας την ψευδαίσθηση ότι βλέπουμε μέσα από αυτή.
- Τη λύση στο πρόβλημα των πολλαπλών διαδοχικών μετασχηματισμών και της εξομοίωσης της κάμερας έρχεται να δώσει η εντολή **gluLookAt()** η οποία μας επιτρέπει να δούμε εύκολα μία 3Δ σκηνή από μία συγκεκριμένη οπτική γωνία.



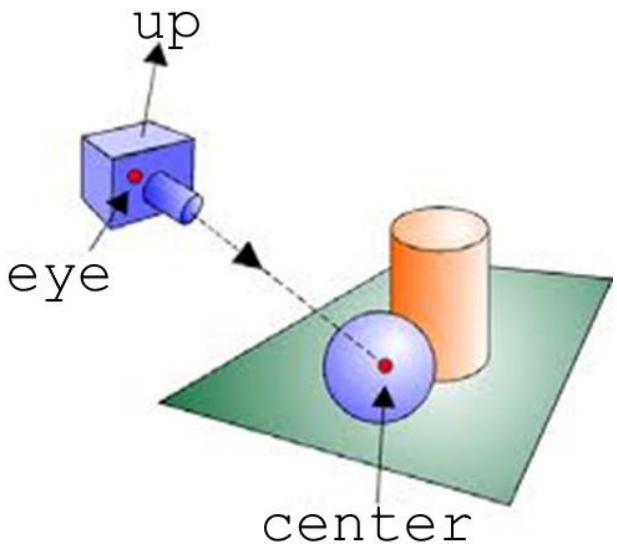


Κάμερα

- Η εντολή **gluLookAt**() συντάσσεται ως εξής:

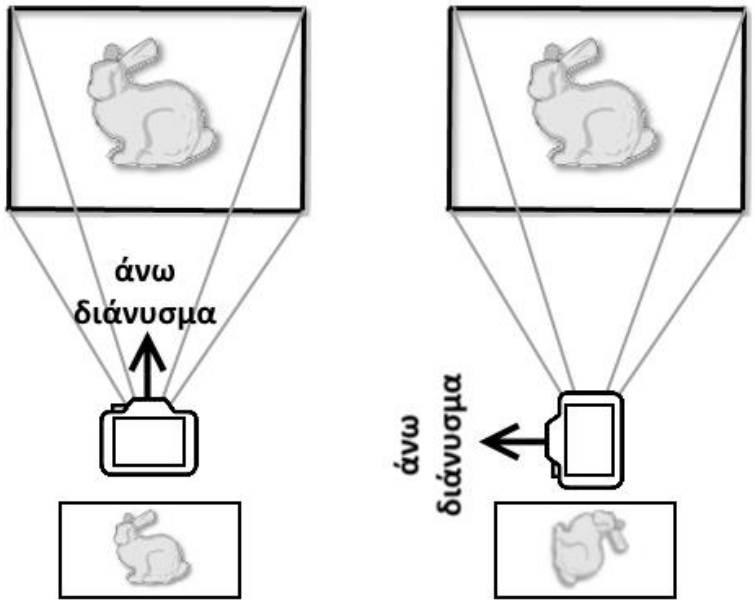
```
gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

eyeX, eyeY, eyeZ: Η θέση που βρίσκεται η κάμερα στον χώρο (ή αλλιώς το μάτι του παρατηρητή).
centerX, centerY, centerZ: Το σημείο στον χώρο στο οποίο η κάμερα εστιάζει.
upX, upY, upZ: Η κατεύθυνση του «άνω διανύσματος» της κάμερας



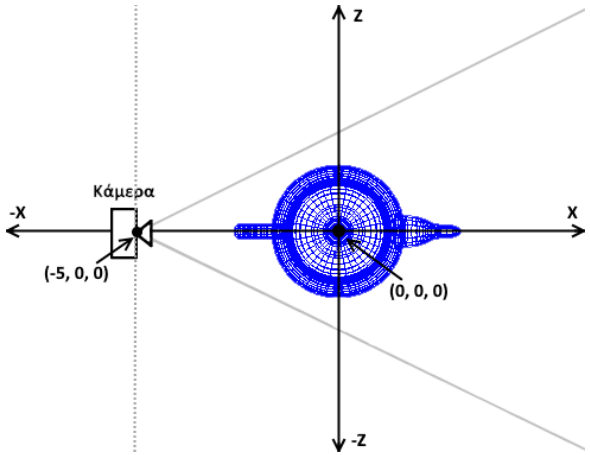
Άνω διάνυσμα της κάμερας

- Η θέση της κάμερας στον χώρο καθώς και το σημείο που αυτή εστιάζει δεν αρκούν από μόνα τους για να αντιληφθούμε τι ακριβώς «βλέπει» η κάμερα. Χρειάζεται και το άνω διάνυσμα αυτής.
- Για να καταλάβουμε τη χρησιμότητα του άνω διανύσματος θα μπορούσαμε να το παρομοιάσουμε ως μία «κεραία» που βρίσκεται πάντα στο επάνω μέρος της κάμερας.
- Από την κλίση της κεραίας μπορούμε να καταλάβουμε την κλίση της κάμερας άρα και το πώς η κάμερα λαμβάνει με το φακό της το είδωλο.
- Επειδή σε ένα πολύ μεγάλο ποσοστό περιπτώσεων η κεραία αυτή βρίσκεται πάντα σε κατακόρυφη θέση, οι συνηθέστερες τιμές του άνω διανύσματος είναι: **upX=0**, **upY=1**, **upZ=0.**

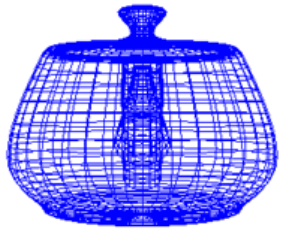


Κοιτώντας την τσαγιέρα από το πλάι με την gluLookAt()

- Ας επιστρέψουμε στην τσαγιέρα. Έστω ότι θέλουμε να δούμε την τσαγιέρα ξανά από το πλάι (αριστερά αυτής, στο χερούλι) με το που ξεκινάει το πρόγραμμά μας χρησιμοποιώντας μόνο τη συνάρτηση **gluLookAt()**.
- Αρχικά, θα σταματήσουμε την κίνηση διαγράφοντας από τη **renderGraphics()** τις εντολές που σχετίζονται με τη μετατόπιση και την περιστροφή αυτής καθώς και τις εντολές που επηρεάζουν τη μεταβλητή **angle**.
- Μετά τη φόρτωση του μοναδιαίου πίνακα, εισάγουμε την εντολή **gluLookAt()** με τις ακόλουθες παραμέτρους:

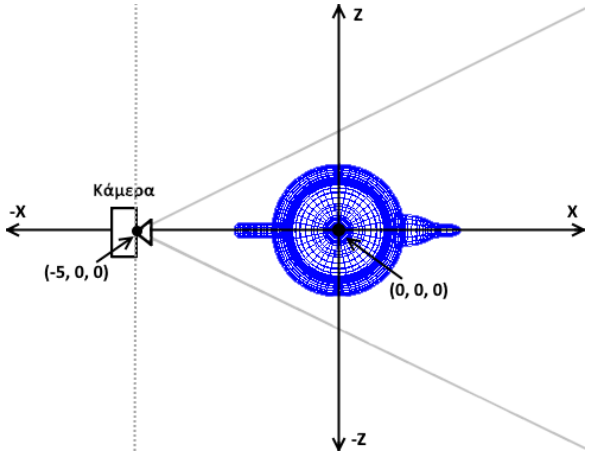



```
void renderGraphics(void)
{
   glClearColor(1, 1, 1, 0);
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();
   //         eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ
   gluLookAt(-5,   0,   0,     0,      0,      0,    0,  1,  0);
   glColor3ub(0, 0, 255);
   glutWireTeapot(1.0);
   glutSwapBuffers();
}
```

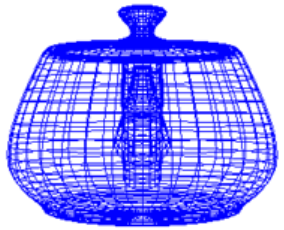


Κοιτώντας την τσαγιέρα από το πλάι με την gluLookAt()

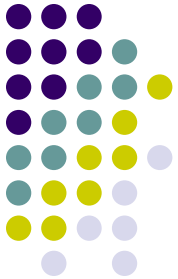
- Παρατηρούμε 2 πράγματα για την **gluLookAt()**:
  1. Δηλώνεται αμέσως μετά την φόρτωση του μοναδιαίου πίνακα και πριν από τη σχεδίαση του αντικειμένου. Άρα, πρώτα «στήνουμε» την κάμερα στη σκηνή και μετά τοποθετούμε τα αντικείμενά μας σε αυτή.
  2. Συνάδει περισσότερο με τη λογική της κάμερας που κινείται ενώ οι άξονες του συστήματος παραμένουν αμετάβλητοι.




```
void renderGraphics(void)
{
   glClearColor(1, 1, 1, 0);
   glClear(GL_COLOR_BUFFER_BIT);
   glLoadIdentity();
   //         eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ
   gluLookAt(-5,   0,   0,     0,      0,      0,    0,  1,  0);
   glColor3ub(0, 0, 255);
   glutWireTeapot(1.0);
   glutSwapBuffers();
}
```

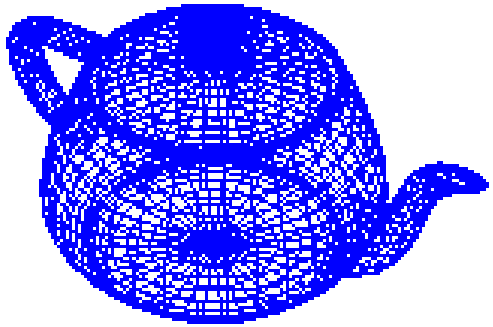


Περιστρέφοντας την τσαγιέρα με την gluLookAt()

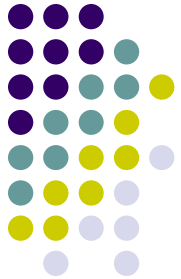


- Το επόμενο πράγμα που θα προσπαθήσουμε να κάνουμε είναι να περιστρέψουμε την κάμερα γύρω από την τσαγιέρα με την **gluLookAt()**.
- Η τσαγιέρα βρίσκεται σταθερή στην αρχή των αξόνων (σημείο {0, 0, 0}) ενώ η κάμερα θα περιστρέφεται γύρω της σε απόσταση 8 μονάδων από αυτή.
- Αυτή τη φορά θα ανυψώσουμε την κάμερα σε απόσταση 6 μονάδων από το «έδαφος» (επίπεδο X-Z, +6 μονάδες στον άξονα Y).
- Για να γίνουν αυτά, δηλώνουμε στην αρχή του προγράμματος τη βιβλιοθήκη **cmath** καθώς και τις μεταβλητές **eyeX, eyeY, eyeZ**.

```
#include <cmath>
#include <GL/glut.h>

float fovy  = 45.0f, angle = 0, eyeX, eyeY, eyeZ;
```

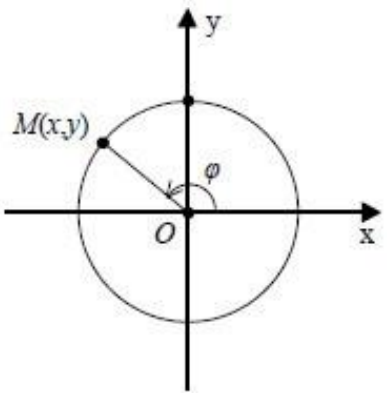


Περιστρέφοντας την τσαγιέρα με την gluLookAt()



- Έστω ο κύκλος *C:* $x^2 + y^2 = \rho^2$ και ένα σημείο *M(x,y)* του καρτεσιανού επιπέδου. Αν το σημείο M(x,y) ανήκει στον κύκλο C και φ∈[0, 2π) είναι η γωνία που σχηματίζει το διάνυσμα OM με τον άξονα x'x , τότε, όπως γνωρίζουμε από την Τριγωνομετρία, θα ισχύουν οι σχέσεις:

  x = ρ · συνφ και **y = ρ · ημφ**

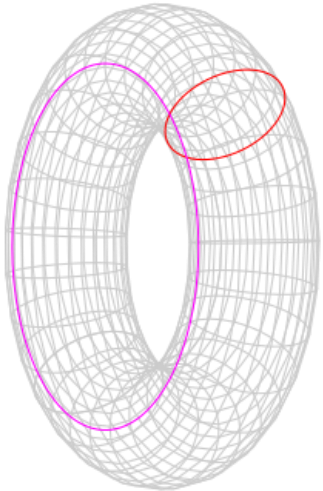


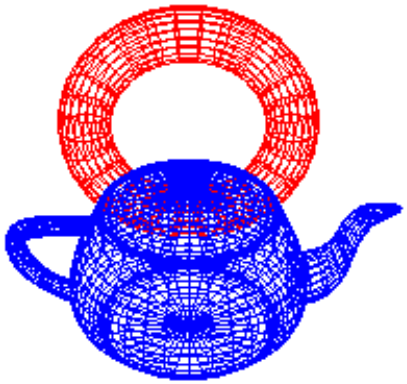
- Εμείς θέλουμε η κάμερα να κάνει κυκλική κίνηση στο επίπεδο X-Z.
- Οπότε, στη συνάρτηση **main()** ενσωματώνουμε τις παραπάνω εξισώσεις στις συντεταγμένες **eyeX** και **eyeZ**
- Οι συναρτήσεις **sin()** και **cos()** δέχονται τη γωνία σε ακτίνια.
- Το **eyeY** θα είναι σταθερά 6

```
void renderGraphics(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    eyeX = 8 * sin(angle * M_PI / 180);
    eyeZ = 8 * cos(angle * M_PI / 180);
    eyeY = 6;
    //          eyeX  eyeY  eyeZ centerX centerY centerZ upX upY upZ
    gluLookAt(eyeX, eyeY, eyeZ,   0,      0,      0,    0,  1,  0);
    angle += 1;
    if(angle > 359) angle = 0;
    glColor3ub(0, 0, 255);
    glutWireTeapot(1.0);
    glutSwapBuffers();
}
```

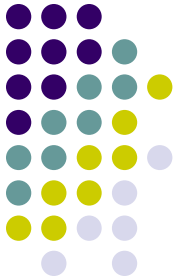
Εισάγοντας έναν τόρο στη σκηνή

- Στη γεωμετρία, ο **τόρος** (torus) είναι ένα στερεό εκ περιστροφής που παράγεται από την περιστροφή ενός κύκλου στον τρισδιάστατο χώρο γύρω από έναν άξονα συνεπίπεδο με τον κύκλο.
- Η λέξη τόρος προέρχεται από την λατινική λέξη torus, που σημαίνει μαξιλάρι. Μερικοί τον αποκαλούν και «λουκουμά».
- Θα εισάγουμε έναν κόκκινο τόρο στη σκηνή μας. Θα τον τοποθετήσουμε 2 μονάδες πίσω από την τσαγιέρα.
- Η τσαγιέρα σχεδιάζεται κάθε φορά στο κέντρο της σκηνής, δηλαδή στο σημείο {0, 0, 0}, οπότε ο τόρος θα σχεδιάζεται στο σημείο {0, 0, -2} αυτής.





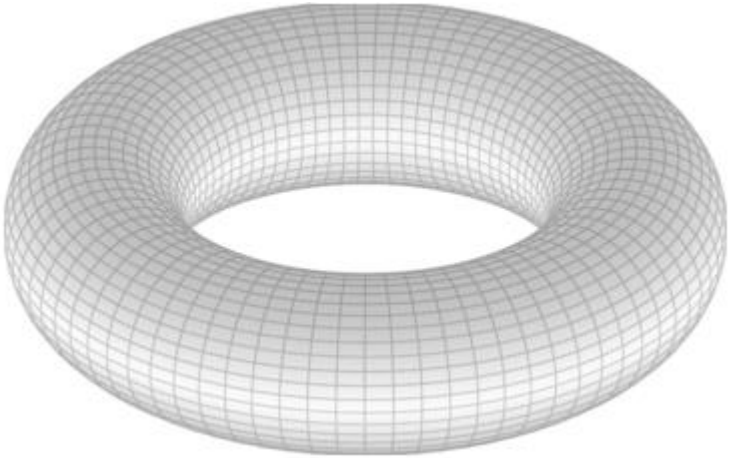
Εισάγοντας έναν τόρο στη σκηνή

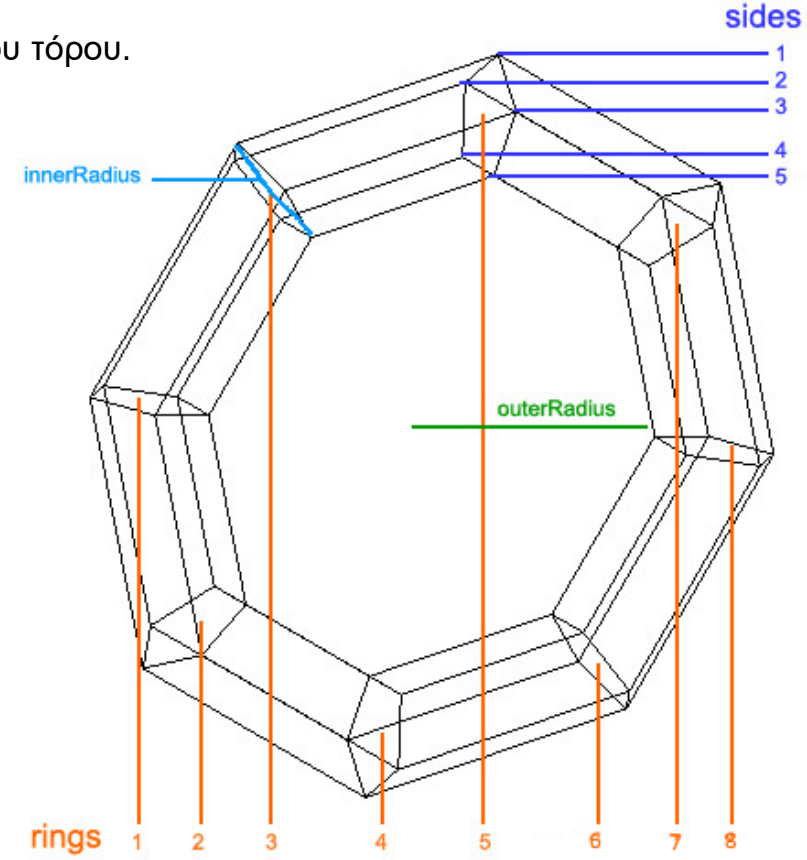


- Είδαμε ότι με τη συνάρτηση **glutWireTorus()** μπορούμε να εισάγουμε έναν έτοιμο τόρο με τη μορφή πλέγματος. Η συνάρτηση δέχεται τις ακόλουθες παραμέτρους:

```
glutWireTorus(innerRadius, outerRadius, nsides, rings)
```

innerRadius: Η ακτίνα κάθε εσωτερικού δακτυλίου του τόρου.
outerRadius: Η εξωτερική ακτίνα του τόρου.
nsides: Ο αριθμός των πλευρών κάθε δακτυλίου.
rings: Ο αριθμός των εσωτερικών δακτυλίων.





Εισάγοντας έναν τόρο στη σκηνή

- Για την εισαγωγή του τόρου δημιουργούμε πρώτα την ακόλουθη συνάρτηση:

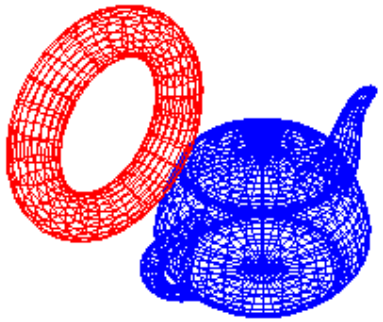
```
void drawTorus()
{
    glColor3ub(255, 0, 0);
    glPushMatrix();
    glTranslatef(0, 0, -2);
    glutWireTorus(0.25, 1, 20, 30);
    glPopMatrix();
}
```

Με την εντολή glPushMatrix() «αποθηκεύουμε» το μητρώο μετασχηματισμών σε μία στοίβα και με την εντολή glPopMatrix() το επαναφέρουμε στην αρχική κατάσταση.

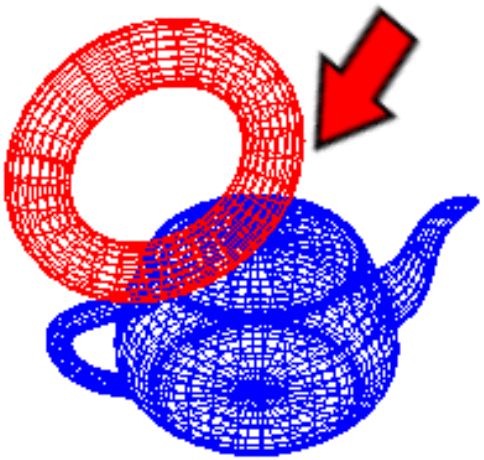
Στο συγκεκριμένο παράδειγμα, θέλουμε από το σημείο που βρισκόμαστε να μεταβούμε στο βάθος κατά 2 μονάδες, να σχεδιάσουμε το αντικείμενό μας και μετά να επιστρέψουμε πίσω χωρίς να επηρεάσουμε το τρέχων σύστημα συντεταγμένων.

- Και στη **renderGraphics()** προσθέτουμε την κλήση της **drawTorus()** λίγο πριν την **glutSwapBuffes()**

```
void renderGraphics(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    eyeX = 8 * sin(angle * M_PI / 180);
    eyeZ = 8 * cos(angle * M_PI / 180);
    eyeY = 6;
    //        eyeX  eyeY  eyeZ centerX centerY centerZ upX upY upZ
    gluLookAt(eyeX, eyeY, eyeZ,   0,      0,      0,    0,  1,  0);
    angle += 1;
    if(angle > 359) angle = 0;
    glColor3ub(0, 0, 255);
    glutWireTeapot(1.0);
    drawTorus();
    glutSwapBuffers();
}
```



Βάθος και προτεραιότητα σχεδίασης

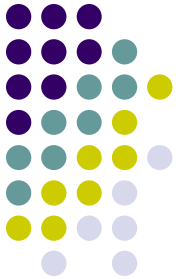


- Αν παρατηρήσουμε προσεκτικά την περιστροφική κίνηση της κάμερας θα διαπιστώσουμε ότι ο κόκκινος τόρος σχεδιάζεται πάντα μπροστά από την τσαγιέρα, αν και είναι τοποθετημένος πίσω της.
- Αυτό συμβαίνει επειδή η OpenGL εξ ορισμού σχεδιάζει τα σχήματα με τη σειρά που της δίνονται οι εντολές.
- Ο προγραμματιστής είναι υπεύθυνος για το ποιο σχήμα θα σχεδιαστεί επάνω σε ποιο.
- Μπορούμε να αλλάξουμε την επιλογή αυτή και να θέσουμε η προτεραιότητα στον σχεδιασμό να γίνεται αυτόματα βάσει του βάθους που βρίσκεται το κάθε αντικείμενο.

- Στη **main()** κάνουμε τις εξής προσθήκες:

```
int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(640, 480);
  glutCreateWindow("3D World");
  glutDisplayFunc(renderGraphics);
  glutReshapeFunc(reshapeWindow);
  glutTimerFunc(1000.0 / 60.0, timer, 0);
  glEnable(GL_DEPTH_TEST);
  glutMainLoop();
  return 0;
}
```

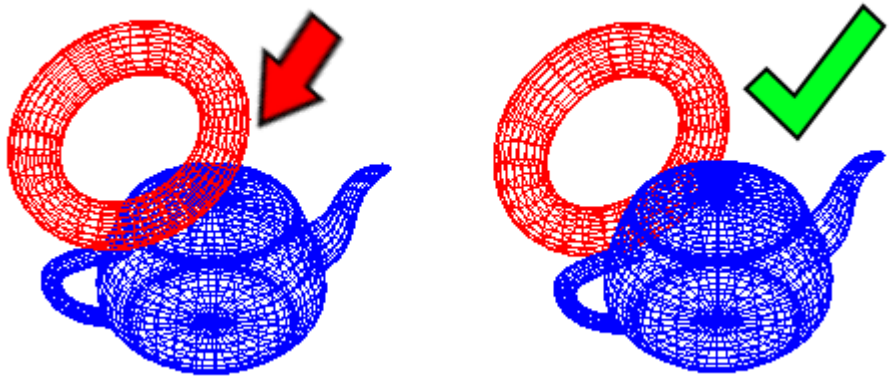
Βάθος και προτεραιότητα σχεδίασης



- Και στη **renderGraphics()** :

```
void renderGraphics(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    eyeX = 8 * sin(angle * M_PI / 180);
    eyeZ = 8 * cos(angle * M_PI / 180);
    eyeY = 6;
    //          eyeX  eyeY  eyeZ centerX centerY centerZ upX upY upZ
    gluLookAt(eyeX, eyeY, eyeZ,   0,      0,      0,    0,  1,  0);
    angle += 1;
    if(angle > 359) angle = 0;
    glColor3ub(0, 0, 255);
    glutWireTeapot(1.0);
    drawTorus();
    glutSwapBuffers();
}
```

- Αν εκτελέσουμε ξανά το πρόγραμμά μας, το πρόβλημα που υπήρχε με το βάθος διορθώθηκε.



Έδαφος

- Με την ίδια λογική που δημιουργήσαμε τον τόρο μπορούμε να σχεδιάσουμε και έδαφος για τον 3Δ κόσμο που τοποθετήσαμε τα αντικειμένά μας.
- Δημιουργούμε μία νέα συνάρτηση με όνομα **drawGround()** και εισάγουμε τις ακόλουθες εντολές:

```
void drawGround()
{
    glColor3ub(200, 200, 200);
    glPushMatrix();
    glScalef(5, 0, 5);
    glBegin(GL_QUADS);
        glVertex3f(-1, 0, 1);
        glVertex3f(1, 0, 1);
        glVertex3f(1, 0, -1);
        glVertex3f(-1, 0, -1);
    glEnd();
    glPopMatrix();
}
```

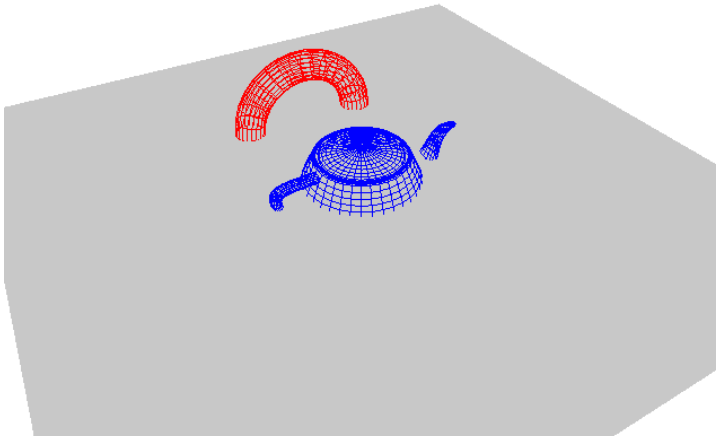
Θέλουμε το έδαφος να έχει διαστάσεις 10x10 (από -5 έως 5) στις διαστάσεις X και Z. Για το λόγο αυτό μεγεθύνουμε προσωρινά τους άξονες X και Z επί 5 μονάδες.

Εναλλακτικά, χωρίς τη χρήση της glScalef(), θα μπορούσαμε να γράψουμε:

```
glVertex3f(-5, 0, 5);
glVertex3f(5, 0, 5);
glVertex3f(5, 0, -5);
glVertex3f(-5, 0, -5);
```

- Στη **renderGraphics()** προσθέτουμε την κλήση της **drawGround()**:

```
    . . . . . .
    glutWireTeapot(1.0);
    drawTorus();
    drawGround();
    glutSwapBuffers();
}
```



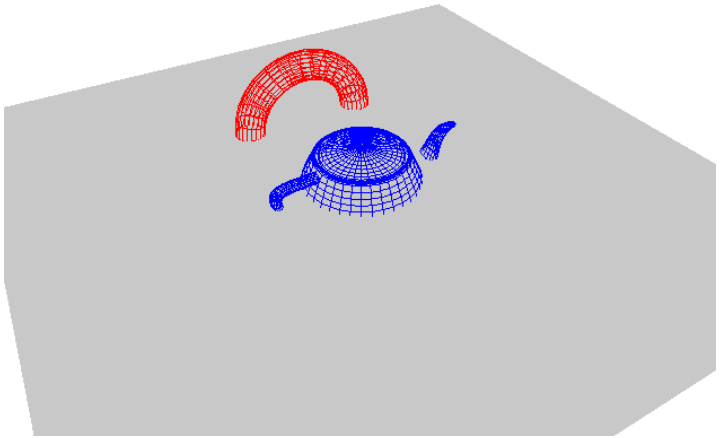
Έδαφος

- Το έδαφος «κόβει» τα αντικείμενα στη μέση. Αυτό είναι λογικό γιατί όλα σχεδιάστηκαν με τη διάσταση Y να είναι 0.
- Όσον αφορά τον τόρο, αυτό διορθώνεται εύκολα αλλάζοντας τη διάσταση Y στην **glTranslatef()** που υπάρχει στην **drawTorus()**.

```
void drawTorus()
{
    glColor3ub(255, 0, 0);
    glPushMatrix();
    glTranslatef(0, 1.25, -2);
    glutWireTorus(0.25, 1, 20, 30);
    glPopMatrix();
}
```

- Όσον αφορά την τσαγιέρα, θα δημιουργήσουμε τη δική της συνάρτηση:

```
void drawTeapot()
{
    glColor3ub(0, 0, 255);
    glPushMatrix();
    glTranslatef(0, 1, 0);
    glutWireTeapot(1.0);
    glPopMatrix();
}
```

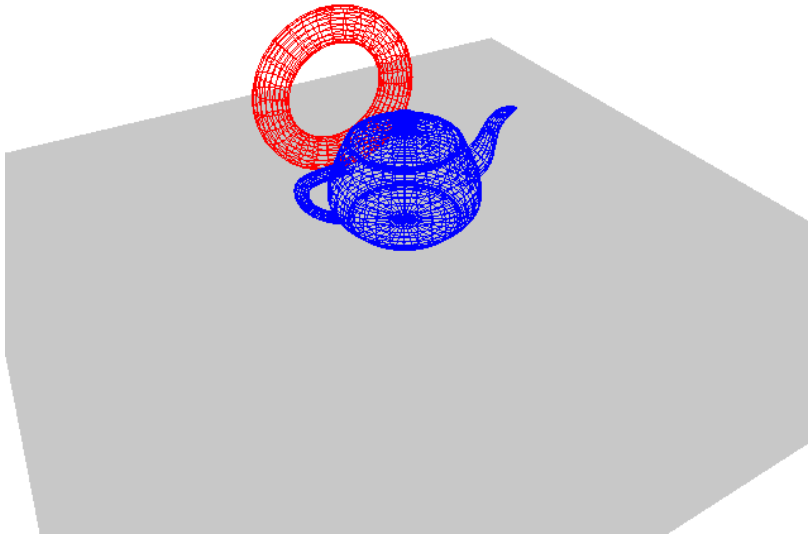


Έδαφος

- Η **renderGraphics()** τώρα αλλάζει σε:

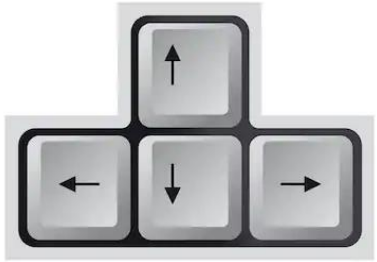
```
void renderGraphics(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    eyeX = 8 * sin(angle * M_PI / 180);
    eyeZ = 8 * cos(angle * M_PI / 180);
    eyeY = 6;
    //         eyeX  eyeY  eyeZ centerX centerY centerZ upX upY upZ
    gluLookAt(eyeX, eyeY, eyeZ,   0,      0,      0,    0,  1,  0);
    angle += 1;
    if(angle > 359) angle = 0;
    drawTeapot();
    drawTorus();
    drawGround();
    glutSwapBuffers();
}
```

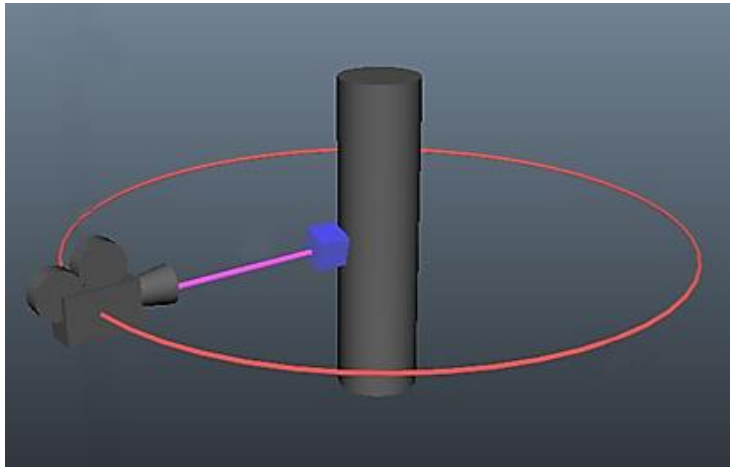
- Το πρόβλημα με το έδαφος διορθώθηκε.



Κίνηση της κάμερας με το πληκτρολόγιο

- Θα χρησιμοποιήσουμε τα βέλη του πληκτρολογίου για να κινήσουμε την κάμερα στη σκηνή.
- Το πάνω και το κάτω βέλος θα μεταβάλλουν το ύψος της κάμερας (θα επηρεάζουν τη μεταβλητή **eyeY**).
- Το δεξί και αριστερό βέλος θα στρέφουν την κάμερα γύρω από τη σκηνή (θα επηρεάζουν τη μεταβλητή **angle**).





Κίνηση της κάμερας με το πληκτρολόγιο

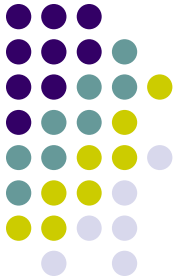
- Δηλώνουμε στο τμήμα δηλώσεων καθολικών μεταβλητών (στην αρχή του κώδικα) δύο πίνακες λογικών τιμών:

```
bool* keyNormal = new bool[256];
bool* keySpecial = new bool[256];
```

- Στη **main()**, μετά τη δήλωση του χρονιστή, προσθέτουμε τα γεγονότα για τον έλεγχο όλων των πλήκτρων:

```
glutKeyboardFunc(normalKeyPressed);
glutKeyboardUpFunc(normalKeyReleased);
glutSpecialFunc(specialKeyPressed);
glutSpecialUpFunc(specialKeyReleased);
```

Κίνηση της κάμερας με το πληκτρολόγιο



- Προσθέτουμε τις αντίστοιχες συναρτήσεις για κάθε ένα γεγονός, κάπου έξω από τη **main()**:

```
void normalKeyPressed(unsigned char key, int x, int y){ keyNormal[key] = true; }
void specialKeyPressed(int key, int x, int y){ keySpecial[key] = true; }
void normalKeyReleased(unsigned char key, int x, int y) { keyNormal[key] = false; }
void specialKeyReleased(int key, int x, int y){ keySpecial[key] = false; }
```

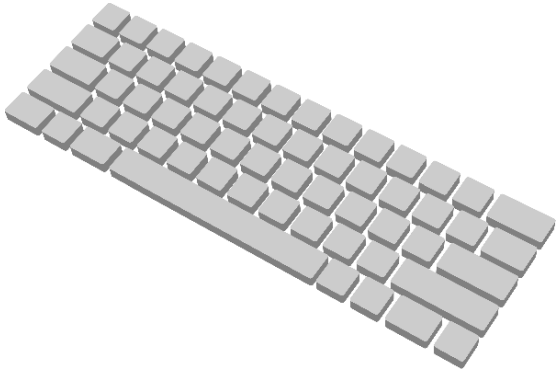
- Επιπλέον, έξω από τη **main()**, δημιουργούμε και μία νέα συνάρτηση με το όνομα **checkKeyboard()** στην οποία θα μπουν όλες οι ενέργειες που σχετίζονται με τα πατήματα των πλήκτρων.

```
void checkKeyboard()
{


}
```

Κίνηση της κάμερας με το πληκτρολόγιο

- Προσθέτουμε την κλήση της **checkKeyboard()** στη **main()**. Διαγράφουμε από αυτή τις μεταβλητές **eyeY** και **angle** :

```
void renderGraphics(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    checkKeyboard();
    eyeX = 8 * sin(angle * M_PI / 180);
    eyeZ = 8 * cos(angle * M_PI / 180);
    //        eyeX  eyeY  eyeZ centerX centerY centerZ upX upY upZ
    gluLookAt(eyeX, eyeY, eyeZ,   0,      0,      0,    0,  1,  0);
    drawTeapot();
    drawTorus();
    drawGround();
    glutSwapBuffers();
}
```

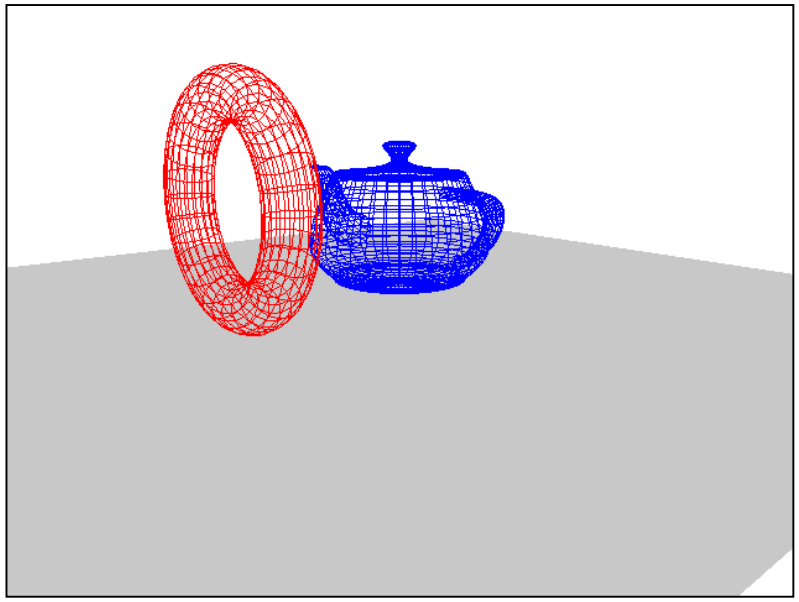


Κίνηση της κάμερας με το πληκτρολόγιο

- Τέλος, προσθέτουμε τον παρακάτω κώδικα στην **checkKeyboard()**

```
void checkKeyboard()
{
    if (keySpecial[GLUT_KEY_UP])
    {
        eyeY += 0.5;
        if(eyeY > 20) eyeY = 20;
    }

    if (keySpecial[GLUT_KEY_DOWN])
    {
        eyeY -= 0.5;
        if(eyeY < -20) eyeY = -20;
    }

    if (keySpecial[GLUT_KEY_LEFT])
    {
        angle -= 1;
        if(angle < 0) angle = 359;
    }

    if (keySpecial[GLUT_KEY_RIGHT])
    {
        angle += 1;
        if(angle > 359) angle = 0;
    }
}
```

Προαιρετικά βάζουμε όριο στο ύψος που μπορεί να ανέβει η κάμερα (από -20 έως 20 μονάδες στον άξονα Y).



Τερματισμός με το πλήκτρο ESC

- Πολλές φορές υπάρχει η ανάγκη να τερματίσουμε το παιχνίδι μας πατώντας απλώς ένα πλήκτρο.
- Το πλήκτρο αυτό είναι συνήθως το **ESC**.
- Στον πίνακα ASCII ο χαρακτήρας **ESC** βρίσκεται στη θέση 27. Άρα, το μόνο που χρειάζεται να κάνουμε είναι να ελέγξουμε αν το πλήκτρο με τον κωδικό 27 πατήθηκε.
- Για να τερματίσουμε το πρόγραμμά μας χρησιμοποιούμε τη συνάρτηση **exit()** της C++.
- Οπότε, στην **checkKeyboard()** τοποθετούμε τις επόμενες γραμμές:

```
// quit when ESC is pressed
if(keyStates[27]) exit(0);
```

ΤΕΛΟΣ