OPENGL შესავალი (2 ლექცია)

OPENGL არის (Application Programming Interface) გრაფიკული პროგრამირების სტანდარტული API (www.opengl.org). პირველი სპეციფიკაცია შეიქმნა 1992 წელს. პირველი იმლემენტაცია 1993 წელს. OPENGL-ის ევოლუცია კონროლირდება ARB (Architecture Review Board)-ის მიერ 1992-წლიდან. 2006 წლის სექტემბრიდან ARB გახდა OPENGL Working Group და შეუერთდა Kronos Groups-ს (www.kronos.org). ARB-ს წევრები არიან 3Dlabs, Apple, ATI, Dell, IBM, Intel, NVIDIA, SGI, Sun Microsystems.

1992 წლის (1.0 რევიზიის) შემდეგ გამოვიდა OPENGL-ეს 6 რევიზია. მიმდინარე რევიზიის ნომერია 2.0.

ვერსია 1.1 - დასრულდა 1997 წელს და დაემატა 2 მნიშვნელოვანი ცვლილება: ვერტექსების(წვეროების) მასივი და ტექტურირებული ობიექტები.

ვერსია 1.2 - გამოვიდა 1998 წელს, დაემატა 3D ტექსტურების მხარდაჭერა და 2D ნახატებთან მუშაობის შესაძლებლობა.

ვერსია 1.3 - გამოვიდა 2001 წელს, დაემატა კუბური, შეკუმშული ტექტურების მხარდაჭერა. მულტიტექტურირება და სხვა.

ვერსია 1.4 - გამოვიდა 2002 წელს, დაემატა ავტომატური მიფ-დონეების მხარდაჭერა, უფრო მეტი ბლენდინგის ფუნქციები, ჩრდილების და მოქნილი ფორმატები და ბევრი სხვა სიახლე.

ვერსია 1.5 - გამოვიდა 2003 წელს, დაემატა ვერტექს ბუფერული ობიექტების მხარდაჭერა, ჩრდილების შედარების ფუნქცია, occlusion queries და სხვა.

1.5-ის ჩათვლით ყველა ვერსიაში მკაცრად იყო განსაზღვრული ფუნქციების თანმიმდევრობა. პროგრამისტს შეეძლო მხოლოდ პარამეტრების შეცვლა. OPENGL 2.0 სპეციფიკაცია დასრულდა 2004 წლის სექტემბერში, და შემოიტანა ვერტექსების და ნაწილების დამუშავების ცნება. ამის შემდგომ პროგრამისტებს თვითონ შეეძლოთ თავისი პროგრამების წერა მაღალი დონის შეიდერულ ენაზე და სხვადასხვა ეფექტების და საკუთარი რენდერირების ალგორითმების დაპროგრამება. ამიტომ ითვლება რომ 1.5-დან 2.0 გადასვლა ეს იყო პრინციპული ცვლილება. მიუხედავად ამისა 2.0 არ დაუკარგავს უკან თავსებადობა 1.5-თან. 2.0-ში ასევე დაემატა რამოდენიმე ბუფერის ერთდოულდ რენდერირების საშუალება, არა-2-ის-ხარისხის ტექსტურების მხარდაჭერა, წერტილოვანი სპრაიტების მხარდაჭერა და სხვა.

შესრულების მოდელი

OPENGL API რენდერისთვის საჭირო მონაცემებს იღებს კადრის(frame) ბუფერიდან. კადრის ბუფერის დიზაინი ისეა მოწყობილი რომ იგი შეიძლება შეიცავდეს ინფორმაციას 3D ობიექტებზეც და ნახატებზეც.

სიმარტივისთვის OPENGL-ი შეიძლება განვიხილოთ როგორც კლიენტ-სერვერული აპლიკაცია. კლიენტი-პროგრამა გარკვეული თანმიმდევრობით აწვდის OPENGL-ის იმლემენტაციას, სხვადასხვა ბრძანებებს. OPENGL-ის ბრძანებები ყოველთვის სრულდება იმ თანმიმდევრობით რა თანმიმდევრობითაც მიეწოდება.

კადრის ბუფერი

OPENGL არის გრაფიკის სახატავი API. აქედან გამომდინარე მისი მთავარი დანიშნულებაა გარდაქმნას მიწოდებული მონაცამები ისე რომ ეკრანზე რაღაც მაინც გამოჩნდეს. ამ პროცესს ხშირად უწოდებენ რენდერინგს. როგორც წესი ეს პროცესი სრულდება გრაფიკული ამაჩქარებლის დახმარებით, თუმცა შეიძლება იგივე პროცესორის ხარჯზეც გაკეთდეს.

თითოეული პიქსელი ეკრანზე წარმოდგინდება როგორ 3 ფერის ერთობლიობა (RGB) და ამ მეხსიერებას ეწოდება დისპლეის მეხსიერება. ეს მეხსირება წამში რამოდენიმეჯერ გადახალისდება ხოლმე და ამ პროცესს უწოდებენ ”რეფრეშს”. არსებობს ე.წ. ”ოფსკინ” მეხსიერებაც რომელიც გამოიყენება იმ მონაცემების შესანახად რომელიც ამწუთას არ ჩანს მონიტორზე.

OPENGL-ი ორივე ტიპის მეხსიერებაზე ”ზრუნვას” ოპერაციულ სისტემას აბარებს. გრაფიკული მეხსიერების იმ ნაწილს, რომელიც იცვლება OPENGL-ის რენდერირების პროცესის დროს, ეწოდება კადრის ბუფერი. მაგალითად, ვინდოუსში ასეთს წარმოადგენს ფანჯარა, სხვა სისტემებში შეიძლება იყოს მთელი ეკრანი.

ფანჯარა რომელიც მხარს OPENGL-ის რენდერირებას შეიძლება ქონდეს შემდეგი ბუფერები:

- 4-მდე ფერის ბუფერი (წინა ბუფერი, უკანა ბუფერი,...)
- სიღრმის (depth) ბუფერი
- სტენსილ ბუფერი
- აკუმულაციის ბუფერი
- მულტისამპლირების ბუფერი
- 1 ან რამოდენიმე აუქსიალური ბუფერი

თანამედროვე გრაფიკული ამაჩქარებელები მხარს უჭერენ წინა და უკანა ბუფერს. ეს გამოიწენება ანიმაციისას ე.წ. ორმაგი ბუფერირება (double buffering). 1 ბუფერიან ფანჯარს აქვს მხოლოდ 1 ფერის ბუფერი - წინა.

როდესაც იხატება 3D ობიექტი, აუცილებლად გვჭირდება სიღრმის ბუფერი, სადაც თითოეული პიქსელიზე სიღრმისეული ინფორმაცია ინახება. როდესაც ახალი ობიექტი ემატება სცენას, შენახული ინფორმაციის საფუძველზე შესაძლებელია შედარების გაკეთება. ამ შედარების შემდეგ ახალი ობიექტის მხოლოდ ის ნაწილები დაიხატება რომლებიც გაივლიან სიღრმისეულ ტესტს.

სტენსილ ბუფერი გამოიყენება რთული მასკირების ოპერაციებისთვის.

დაგროვების ბუფერი არის ფერის ბუფერი. აქ შეიძლება რამოდენიმე ნახატის ერთდოულად დადება ახლის მიღების მიზნით. მაგალითად აქ შეიძება ე.წ. ”მოშენ ბლურ” ეფექტის მიღება. ამ დროს რამოდენიმე კადრის დახატვა ხდება დაგროვების ბუფერში და პიქსელის ფერი იყოფა კადრების რაოდენობაზე. ასევე შეიძლება depth of field ეფექტის მიღებაც. აქვე შეიძლება მაღალი დონის მთელ-ეკრანული ანტიალიასინგის გამოთვლაც.

როცა ეკრანზე იხატება ობიექტი სისტემა იღებს გადაწყვეტილებას თუ როგორ უნდა შეცვალოს კონკრეტული პიქსელის ფერი ეკრანზე. მულტისამპლირების ბუფერი საშულებას გვაძლევს ყველაფერი რაც კი რენდერირდება ეკრანზე რამოდენიმეჯერ გავიმეოროთ უფრო კარგი ხარისხის მიღწევის მიზნით და ეს ხდება სცენის რამოდენიმეჯერ რენდერირების გარეშე. თითოეული სემპლი პიქსელის

შიგნით შეიცავს ინფორმაციას ფერის, სიღრმის, სტენსილის და სემპლირების რაოდენობას. ობიექტის რენდერირების დროს საბოლოოდ მიიღება 1 ფერი რომელიც იწერება ფერის ბუფერში. იმის გამო რომ მულტისემპლირების ბუფერი შეიცავს ბევრ ფერის, სიღრმის, სტენსილის სემპლებს (4,8 ან 16) , თითოეული პიქსელისათვის ეკრანზე, ის ბევრ გრაფიკულ მეხსიერებას იკავებს.

აუქსიალური ბუფერი, ესაა სხვადასხვა მონაცემების შესანახი მეხსიერება, რომელიც არ ჩანს ეკრანზე. აქ შეიძლება სხვადასხვა საშუალედო მონაცემების შენახვა. კადრის ბუფერს შეიძლება ქონდეს 1,2,3,4 ან მეტი აუქსიალური ბუფერი.

მდგომარეობა

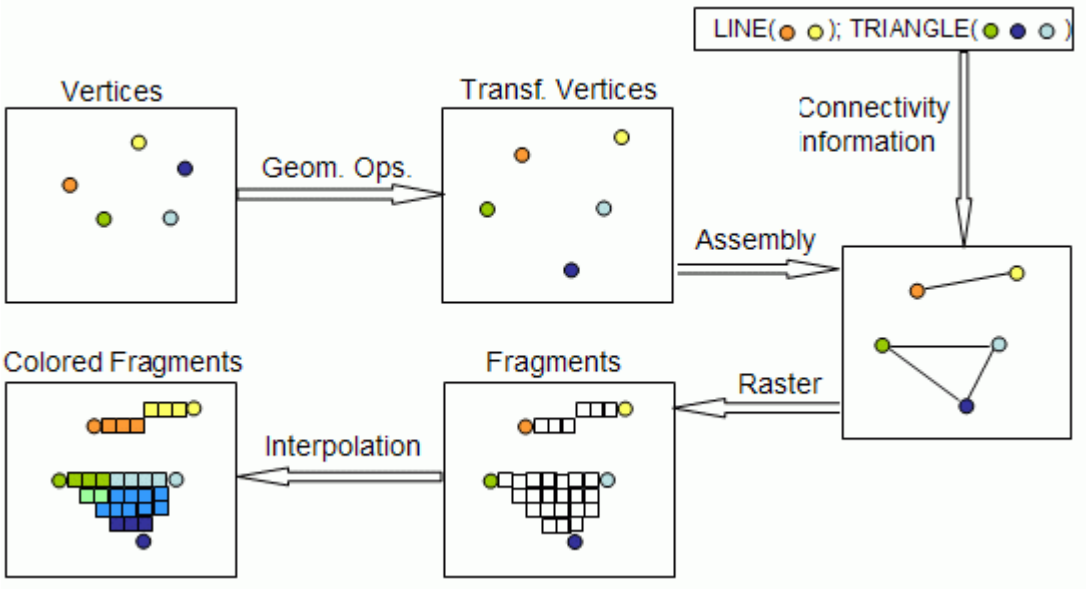
OpenGL-ი თავიდანვე შექმნილი იყო როგორც მდგომარეობის შემნახველი მანქანა, რომლიც მუდმივად განაახლებს კადრის ბუფერს. გეომეტრიული პრიმიტივების, ნახატების და ა.შ. გარდაქმნის პროცესს, წინ უძღვის დიდი რაოდენობის მდგომარეობების შეცვლა/დაყენება. OpenGL-ის მდგომარეობები ქმნიან ერთიან მონაცემთა სტრუქტურას - ”გრაფიკულ კონტექსტს”.

უმარტივესი მდგომარეობების დაყენება ხდება glEnable და გაუქმება glDisable ბრძანებით. მაგალითად glEnableClientState და შესაბამისად glDisableClientState.

OpenGL-ი ასევე უზრუნველყოფს მდგომარებობების სტეკში ჩადება/ამოღებას შემდეგი ბრძანებებით: glPushAttrib და glPopAttrib. მაგალითად : glPushClientAttrib და glPopClientAttrib.

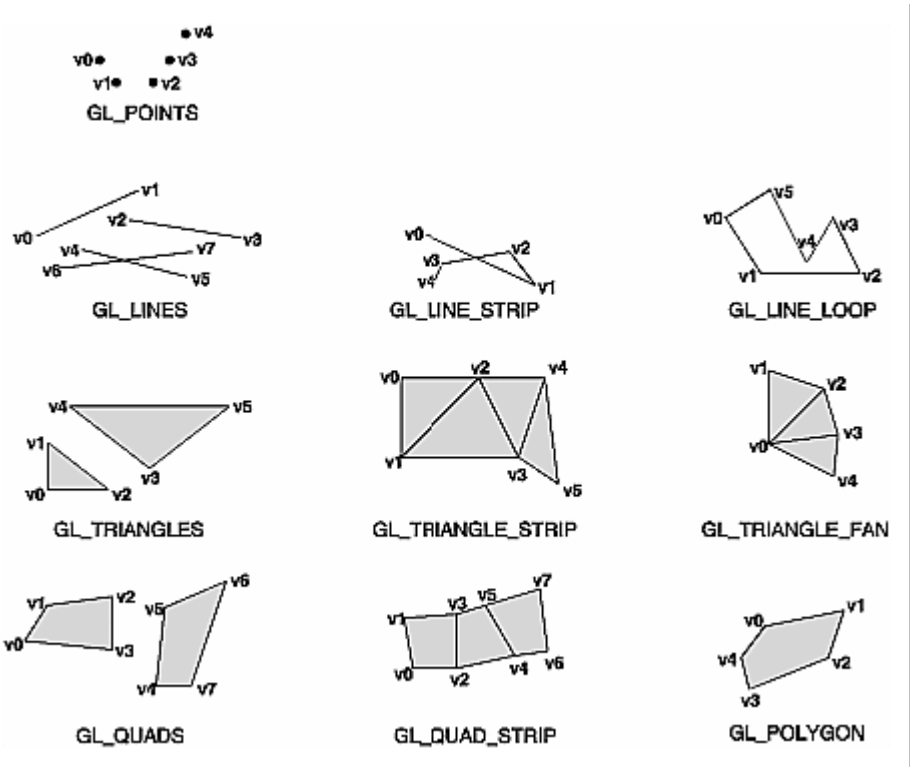
glGet გამოიყენება დაყენებული მდგომარების მონაცემების წასაკითხად. glGetError გამოიყენება შეცდომის დასადგენად.

OpenGL Rendering pipeline



1. გეომეტრიული ინფორმაციის შევსება

OpenGL-ი მხარს უჭერს შემდეგ გეომეტრიულ პრიმიტივებს: წერტილები, ხაზები, ხაზების მიმდევრობა, ხაზების ციკლი, პოლიგონები, სამკუთხედები, სამკუთხედ-ფანები, სამკუთხედ-სტრიპები, კვადრიტერალები, კვადრიტერალ-სტრიპები.



OpenGL-ში განსაზღვრულია, გეომეტრიული მონაცემების მიწოდების 3 ხერხი:

1. 1-წერტილი-დროის-1-მომენტში: იწყება glStart ბრძანების გამოძახებით და მთავრდება glEnd-ით. მათ შორის იწერება სხვადასხვა ბრძანებები რომლებიც აღწერენ წერტილებს, როგორიცაა: პოზიცია, ფერი, ნორმალი, ტექსტურის კოორდინატები, მეორადი ფერი, წვეროს სასაზღვრო მნიშვნელობა, ნისლის(fog) კოორდინატები. ამისთვის გამოიყენება glVertex, glColor, glNormal, glTexCoord და ა.შ. 1.5-დე არ არსებობდა საშუალება წვეროს თან გაყოლოდა რამე დამატებითი ინფორმაცია. 2.0-ში უკვე ამის გაკეთებაც შეიძლება. სამკუთხედების ხატვისას იგულისხმება რომ ყოველი მესამე glVertex-ის გამოძახება არის ახალი სამკუთხედი. ხაზის ხატვისას ყოველი მეორე და ა.შ.

(მაგალითი 1 : Hello world - OpenGL აპლიკაცია)

```
#include "glut.h"

void display(void) {

	glClear(GL_COLOR_BUFFER_BIT);

	glColor3f (1.0, 1.0, 1.0);

	glBegin(GL_POLYGON);

	glVertex3f(0.25, 0.25, 0.0);
	glVertex3f(0.75, 0.25, 0.0);
	glVertex3f(0.75, 0.75, 0.0);
	glVertex3f(0.25, 0.75, 0.0);

	glEnd();
	glFlush ();
}

void init (void) {

	glClearColor(0.0, 0.0, 0.0, 0.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

int main(int argc, char** argv)
	{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowSize(250, 250);
	glutInitWindowPosition(100, 100);
	glutCreateWindow("OpenGL 01");
	init();
	glutDisplayFunc(display);
	glutMainLoop();

	return 0;
}
```

2. მეორე მეთოდი : გეომერტიული მონაცემები ინახება მომხმარებლის მიერ წინასწარ განსაზღვულ მასივებში. შემდეგი ფუნქციების გამოყენებით:
   glDrawArrays, glMultiDrawArrays, glDrawElements, glMultiDrawElements, glDrawRangeElements, ან glInterleavedArrays ხდება ბევრი პრიმიტივების ერთდროულად გამოტანა ეკრანზე. ამ დროს ხდება ერთდროულად ბევრი გეომეტროული ინფორმაციის მიწოდება OpenGL-თვის. ამ ფუნქციების გამოყენება რეკომენდირებულია და გამართლებულია პროგრამის კრიტიკულ ნაწილებში. წინა მეთოდისგან განსხვავებით, როდესაც თითოეული პრიმიტივის დახატვის წინ საჭირო იყო glBegin, glEnd-ი, აქ ესეთი ხარჯები არაა. OpenGL-ში ასეთი ფორმით მიწოდებულ მონაცემები უფრო ადვილად და ეფექტურად მუშავდება.

3. წინა 2 მეთოდის გამოყენების დროს სისტემა იმყოფებოდა ე.წ. IMMEDIATE_MODE-ში. როგორც სახელიდან გამომდინარეობს, როგორც კი განისაზღვრებოდა პრიმიტივები, იქვე ხდებოდა მათი რენდერი. მესამე მეთოდის გამოყენებით შეიძლება პრიმიტივების აღწერა ან 1,2 მეთოდით და მათი შენახვა ე.წ. "DISPLAY_LIST"-ში. ეს არის OpenGL-ის მონაცემთა სტრუქტურა, რომელიც შეიძლება შეიცავდეს სხვადასხვა ბრძანებებს და მდგომარეობებს, რომელებიც მოგვიანებით შესრულდება. ეს მონაცემები ინახება სერვერის მხარეს და მათი შესრულება შეიძლება მოხდეს glCallList და glCallLists ბრძანებების გამოძახების საშუალებით. ახალი სტუქტურის შექმნა ხდება glNewList ბრძანებით, და მთავრდება glEndList-ით. მათ შორის მოთავსებული, ყველა ბრძანება ავტომატურად ხდება DISPLAY_LIST-ში. ამ მეთოდის გამოყენებით შესაძლებელია წარმადობის გაზრდაც. განსაკუთრებით ეფექტურია იმ შემთხვევაში როდესაც კონკრეტული DISPLAY_LIST-ი მოთავსებულია გრაფიკული ადაპტერის მეხსიერებაში და მისი გამოძახება ხდება რამოდენიმეჯერ.

OpenGL-ის 1.5 ვერსიაში დამატებული იქნა ახალი ფუნქციები, რომლებიც საშუალებას გვაძლევს წვეროების შესახებ ინფორმაცია შევინახოთ სერვერის მხარეზე. ესეც საშუალებას გვაძლევს გაცილებით გავზარდოთ წარმადობა რადგან არ გვიწევს გეომეტრიული ინფორმაციის წინ და უკან გადაცემა და ამით სისტემური სალტის დატვირთვა მცირდება.

OpenGL API-ი ასევე შეიცავს გეომეტრიული მონაცმების სწრაფი გადაცემა-მიღების ფუნქციებსაც. glBindBuffer ქმნის ახალ ბუფერს, და glBufferData, glBufferSubData-ფუნქციებით ხდება ამ ბუფერის მონაცემების შევსება. glMapBuffer ფუნციას საშუალებას გვაძლევს მივიღოთ მიმთითებელი ამ მეხსირებაზე და განვახორციელოთ პირდაპირი წვდომა. გაითვალისწინეთ რომ აუცილებელია glUnMapBuffer ბრძანების გამოძახება, OpenGL-თვის შეცვლილი მონაცმების გადაცემის წინ.

2. ოპერაციები წვეროებზე (ვერტექსებზე)

მიუხედავად იმისა რა მეთოდი იქნა გამოყენებული გეომეტრიული ინფორმაციის გადასაცემად, შემდეგი ეტაპი ყოველთვის არის "წვეროების დამუშავება". ამ დროისთვის წვეროების კოორდინატები უკვე გამრავლებული modelview და projection მატრიცებზე, ასევე გარდაქმნილია ნორმალები და ტექტურების კოორდინატებიც. აქვე ხდება განათების გამოთვლაც და ფერების დადებაც არსებული მატერიალების გათვალისწინებით. ყველა ეს ოპერაცია და მათი მოქმედების თანმიმდევრობა მკაცრადაა განსაზღვრული OpenGL-ის მიერ.

ყველაზე მნიშვნელოვანი ამ ეტაპზე არის გარდაქმნები და განათება (Transformation and Lighting, მოკლედ T&L). ეს ეტაპი სრულიად ავტომატიზირებულია და არ საჭიროებს დეველოფერის ჩარევას. ამ პროცესზე ზემოქმედება შესაძლებელია მხოლოდ, წინასწარ რამოდენიმე მდგომარების შეცვლა/დაყენებით, განათების ჩართვა გამორთვით და განთების მოდელის შეცვლით, ასევე მატერიალების თვისებების შეცვლით და ტრანსფორმაციის

მატრიცების შეცვლით (glMatrixMode, glLoadMatrix, glMutlMatrix, glRotate, glScale, glTranslate).

განათება OpenGL-ში განსაზღრულია სინათლის სხვადასხვა წყაროების მიერ. სინათლის წყაროების რაოდენობა მოცემულ იმლემენტაციაში განისაზღვრება კონსტანტით GL_MAX_LIGHT. ეს მნიშვნელობა შეიძლება წაიკითხოთ glGet ფუნქციის საშუალებით. განათება შეიძლება იყოს: მიმართული(directional), წერტილოვანი(point), ან ფანრის ტიპის(spotlight). შესაძლებელია განათების სხვადასხვა პარამეტრების და ფერების შეცვლა/დაყენებაც, როგორიცაა: პოზიცია,RGBA მნიშვნელობები, ამბიენტის, დიფუზიის და სპელუკარული კომპონენტების შეცვლა, ინტენსივობის და ა.შ. შესაძლებელია თითოეული სინათლის წყაროს ჩართვა/გამორთვა glEnable/glDisable ფუნქციების საშუალებით. განათება თითოეული წვეროსთვის ითვლის პირველად და მეორად ფერებს. მთლიანად განათების პროცესის ჩართვა/გამორთვა ხდება glEnable(GL_LIGTHNING)/ glDisable(GL_LIGHTNING) ბრძანებების საშუალებით. ამ შემთხვევაში პიქსელების ფერი განისაზღვრება უკანასკნელი glColor ბრძანებაში დაყენებული ფერით.

განათების ეფექტის მოქმედება კონკრეტულ ობიექტზე დამოკიდებულია ამ ობიექტის მატერიალზე. მატერიალი ხასიათდება იმ ფერით რომლებსაც ისინი გამოასხივებენ და სპეკულარული, დიფუზიური და გაფანტული ნათებით რომლებსაც ისინი აირეკლავენ. ასევე გასათვალისწინებელია მატერიალის Shininess თვისებაც.

სპეკულარული არეკლვა - იგივე სარკისებული, მაგალითად ლაზერი. ზედაპირის არის სწორი, გაპრიალებული. დაცემის კუთხე = არეკვლის კუთხის

დიფუზიური არეკლვა - ხდება სხვადასხავა მიმართულებით, რადგანაც ზედაპირი არის არასწორი. საპირსპიროა სპლეკულარულის.

გაფანტული არეკლვა - ყველა მხრიდან,აფერდადებს.

გლობალური განათება იცვლება glLightModel ფუნქციის საშუალებით:

- შეიძლება გლობალური გაფანტული ნათების ფერის დაყენება მთლიანი სცენისთვის.
- განათება მოქმედებს კონკრეტულს ხედზე თუ უსასრულოა. გამოიყენება სპეკულარული არეკვლის კუთხეების დათვლისას.
- დაზუსტდეს პოლიგონების განათების პარამეტრები იყოს ცალ-თუ-ორმხრივი.
- დაზუსტდეს ცალკე გამოითვალოს თუ არა სპეკულარული კომპონენტი, რომელიც მოგვიანებით შეიძლება გამოყენებული იქნას.

3. პრიმიტივების აწყობა

წვეროების დამუშავების პროცესის შემდეგ იწყება პრიმიტივების აწყობის ეტაპი. ამ ეტაპზე ხდება წვეროებიდან პრიმიტივების გამოყოფა. წერტილებისთვის 1 წვერო, ხაზისთვის 2, სამკუთხედისთვის 3 და ა.შ. თუ გამოყენებული იქნა მეთოდი 1-წვერო-დროის-ერთ-მომენტში მაშინ პრიმიტივის ტიპი განისაზღვრება glBegin ოპერატორით. წვეროების მასივების გამოყენებისას პრიმიტივის ტიპი გადაეცემა

ხატვის ფუნქციას. ამ ეტაპზე ხდება წვეროებიდან პრიმიტივების ფორმირება და მათი გადაცემა შემდეგ ეტაპზე - პრიმიტივების დამუშავება.

4. პრიმიტივების დამუშავება

პირველი რაც ხდება ამ ეტაპზე ესაა მოჭრა (clipping). ამ დროს ხდება მოცემული პრიმიტივის ზედაპირის შედარება მომხარებლის მიერ დაყენებულ სხვადასხვა სიბრტყეებთან და ხდება ამ პრიმიტივის ან დაშვება ან უარყოფა შემდგომი დამუშავებისთვის. ანუ თუ პრიმიტივი ზედაპირი მთლიანად ჯდება წინასწარ განსაზღვრულ საზღვრებში, რჩება და გადაეცემა შემდეგ ეტაპს. თუ ნაწილობრივ ჯდება ხდება "უსარგებლო" ნაწილის ჩამოჭრა და შემდეგ ეტაპზე მხოლოდ დარჩენილი ნაწილის გადაცემა.

მეორე ოპერაცია ესაა, პერსპექტივის პროექცია. ყოველის წვეროს x,y და z კოორდინატა იყოფა თავისივე ჰომოგენურ კოორდინატზე რათა მოხდეს მათი ფანჯრის კოორდინატთა სისტემაში გადაყვანა.

აქვე ასევე შეიძლება მოხდეს ე.წ. პოლიგონების culling-ი. ანუ თუ ჩართულია glCullFace მდგომარეობა, ხდება ყველა პოლიგონის უარყოფა, რომლის ზედაპირიც შეტრიალებულია სცენის საპირისპიროდ ან პირიქით, დამოკიდებულია პარამეტრებზე.

5. რასტერიზაცია

ამ ეტაპზე ხდება, მოცემულ კადრის ბუფერში, გეომეტრიული პრიმიტივების დაქუცმაცება/დაყვანა უფრო პატარა ნაწილებად - პიქსელებად. ამ პროცესს სხვანაერად ეწოდება რასტერიზაცია. თითოეულ ელემენტს რომელიც მიიღება ამ პროცესის შედეგად ეწოდება ფრაგმენტი.

მაგალითად: მოცემულია ხაზი (განსაზღვრულია 2 წვეროს მიერ), რომელიც მონიტორზე იკავებს 5 პიქსელს, ანუ რასტერიზაციის დროს 2 წვერო გარდაიქმება 5 ფრაგმენტად. ფრაგმენტის ფერს განსაზღვრავს წვეროების ფერი, ტექტურის ფერი და ა.შ. რასტერიზაციის დროს წვეროებს შევსებული აქვთ პირველადი და მეორეადი ფერთა თვისებები. glShadeModel ფუქცია განსაღვრავს თუ როგორ გაფერადდეს პრიმიტივი. SMOOTH_SHADING -ის შემთხვევაში ხდება ფერების ინტერპოლირება. FLAT_SHADING-ის დროს ფერადდება ერთი ფერით (უკანასკნელი წვეროს ფერით).

თითოეულს პრიმიტივს აქვს სხვადასხვა რასტერიზაციის პარამეტრები და თვისებები.

წერტილის სიგანე განისაზრვრება glPointSize ფუნქციით. OpenGL 2.0 დამატებულია ნებისმიერი ფორმის წერტილის ხატვა, ანუ შესაძლებელია წერტილისთვის ფორმის დანიშვნა. ამას point sprite ეწოდება.

ხაზის სიგანის შეცვლა ხდება glLineWidth-ფუნქციის საშუალებით. პუნქტირის შეცვლა glLineStripple-ით. ასევე შესაძლებელია პოლიგონებისთვისაც დაინიშნოს 32X32 "პატერნი" glPolygonStripple ფუნქციის საშუალებით. ალიასინგის სხვადასხვა რეჟიმების დანიშვნა ხდება glEnable-ს საშუალებით (GL_POINT_SMOOTH, GL_LINE_SMOOTH, GL_POLYGON_SMOOTH)

```
#include "stdafx.h"
#include "glut.h"

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
      glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

void init(void){
      glClearColor (0.0, 0.0, 0.0, 0.0);
      glShadeModel (GL_FLAT);
      }
void display(void) {

      int i;
      glClear (GL_COLOR_BUFFER_BIT);
      glColor3f (1.0, 1.0, 1.0);
      glEnable (GL_LINE_STIPPLE);
      glLineStipple (1, 0x0101); /* dotted */
      drawOneLine (50.0, 125.0, 150.0, 125.0);
      glLineStipple (1, 0x00FF); /* dashed */
      drawOneLine (150.0, 125.0, 250.0, 125.0);
      glLineStipple (1, 0x1C47); /* dash/dot/dash */
      drawOneLine (250.0, 125.0, 350.0, 125.0);
      glLineWidth (5.0);
      glLineStipple (1, 0x0101); /* dotted */
      drawOneLine (50.0, 100.0, 150.0, 100.0);
      glLineStipple (1, 0x00FF); /* dashed */
      drawOneLine (150.0, 100.0, 250.0, 100.0);
      glLineStipple (1, 0x1C47); /* dash/dot/dash */
      drawOneLine (250.0, 100.0, 350.0, 100.0);
      glLineWidth (1.0);
      glLineStipple (1, 0x1C47); /* dash/dot/dash */
      glBegin (GL_LINE_STRIP);
      for (i = 0; i < 7; i++)
            glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
      glEnd ();
      for (i = 0; i < 6; i++) {
            drawOneLine (50.0 + ((GLfloat) i * 50.0), 50.0,
                  50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
            }
      glLineStipple (5, 0x1C47); /* dash/dot/dash */
      drawOneLine (50.0, 25.0, 350.0, 25.0);
      glDisable (GL_LINE_STIPPLE);
      glFlush ();
      }

void reshape (int w, int h){
      glViewport (0, 0, (GLsizei) w, (GLsizei) h);
      glMatrixMode (GL_PROJECTION);
      glLoadIdentity ();
      gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
      }

int main(int argc, char** argv){
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize (400, 150);
      glutInitWindowPosition (100, 100);
      glutCreateWindow (argv[0]);
      init ();
      glutDisplayFunc(display);glutReshapeFunc(reshape);
      glutMainLoop();
      return 0;
}
```

6. ფრაგმენტების დამუშავება

ყველაზე მნიშვნელოვანი ოპერაცია ამ ეტაპზე არის ტექსტურირება. ამ დროს ხდება ტექსტურების მეხსიერებაზე მიმართვა ყოველი კონკრეტული ფრაგმენტისთვის, რომელთანაც ასოციაციერებულია ტექსტურის კოორდინატა. OpenGL- ს აქვს ბევრი თვისებები, რომელიც განსაზღვარს თუ როგორ მოხდეს წვდომა ამ მეხსიერებაზე და როგორ მოხდეს მიღებული მნიშვნელობების დადება მოცემულ ფრაგმეტზე. დეტალურად მოგვიანებით.

ტექსტურირების გარდა ამ ეტაპზე ხდება ნისლის ეფექტის გათვლაც.

ასევე ხდება ფერთა აჯამვა. (ფრაგმენტის პირველადი და მეორადი ფერების მიხედვით)

7. ქვე-ფრაგმენტების დამუშავება

ამ ეტაპზე ხდება ხდება თითოეული ფრაგმენტის შემოწმება:

- პიქსელის-მიკუთვნების-ტესტი: ამ დროს მოწმდება ეკუთვნის თუ არა კონკრეტული პიქსელი ფანჯარას თუ გადაფარულია სხვა ფანჯრის მიერ.
- მაკრატლის-ტესტი: ხდება რეგიონის მოჭრა რომელიც წინასწარ განსაზღვრულია glScissor ფუნქციის მიერ.
- ალფა-ტესტი: მოწმდება გათვალისწინებული იქნას თუ არა ფერის ალფა კომპონენტი. (glAlphaFunc)
- სტენსილ-ტესტი: აქ ხდება პიქსელის შედარება სტენსილ ბუფერის შესაბამის მნიშვნელობასთან glStencilFunc და glStencilOp ფუნქციების გამოყენებით. ტესტის საფუძველზე დგინდება მოცემული ფრაგმენტის ”ბედი”.
- სიღრმისეული-ტესტი: აქ ხდება მოცემული ფრაგმენტის სიღრმის შედარება სიღმის ბუფერში დაფიქსირებულ მნიშვნელობასთან glDepthFunc-ის საშუალებით

ყველა ეს ოპერაცია არ ითვლება ”მძიმე” ოპერაციად და თანამედროვე ვიდეო ამაჩქარებლები ადვილად უმკლავდებიან.

კადრის ბუფერის ოპერაციები

არსებობს ზოგი ფუნქცია რომელიც პირდაპირ მართავს და ცვლის კარდის ბუფერს. ასეთებია სტერეო ნახატები, ან ორმაგი ბუფერი. მიმდინარე აცტიური ბუფერის არჩევა ხდება glDrawBuffer ფუნქციის საშუალებით. ასევე შეიძლება ბუფერში ზოგი რეგიონის ჩაწერისაგან დაცვაც. glColorMask ფუნქციის საშუალებით შეიძლება კონკრეტული ფერების ან ალფა კონპონენტის ჩაწერისგან დაცვა. glDepthMask განსაზღვრავს შეიძლება თუ არა მოცემული ბუფერში სიღრმის ინფორმაციის შეცვლა. ანალოგიურად glStencilMask. ბუფერის მონაცემების ინიციალიზაცია ხდება glClear ფუნქციის საშუალებით. glClear-ის მიერ გამოყენებული მნიშვნელობები ყენდება glClearColor, glClearDepth, glClearStencil და glClearAccum ფუნქციების საშუალებით.

glFlush - გამოიყენება მონაცემების გადასაცემად კონკრეტული რენდერინგ კონტექსტისთვის.

glFinish - ახდებს ყველა ოპერაციის ბლოკირებას სანამ მიმიდინარე გრაფიკული კონტექსტის დამუშავება არ მორჩება. ამის გამო შესაძლებელია წარმადობის მკვეთრი შემცირება. ამიტომ ზომიერად უნდა გამოვიყენოთ ეს ფუნქცია.