مقدمه‌ای بر OpenGL در C++

نویسنده : بهرام بهرام‌بیگی

قابل دانلود از : http://www.itnotes.info

OpenGL چیست ؟

مخفف عبارت Open Graphic Library است و همان طور که از نامش پیداست یک کتابخانه جهت دستیابی ساده به سخت افزار گرافیکی شماست. برای کاربران عادی نام OpenGL با نام بازی های سه بعدی گره خورده است. قبل از نوشتن OpenGL برنامه نویسان گرافیکی مجبور بودند که مستقیما با سخت افزارهای گرافیکی کار کنند و برای سخت افزارها برنامه های اختصاصی بنویسند. اما با گسترش روز افزون تکنولوژی و سخت افزارهای مختلف این رویه ی برنامه نویسی کاربرد خود را از دست داد. چرا که برنامه نویسان برای انتقال یک برنامه ی ساده از یک سخت افزار به سخت افزار دیگر مجبور بودند ده ها (بلکه هزاران) خط کد را ویرایش و اصلاح کنند تا با سخت افزار موجود جواب بگیرند. حتی بعد از برطرف شدن مشکل با سیستم عامل های نسل جدید ، مشکل خود را به گونه ای دیگر نشان داد : « عدم توانایی حمل در سیستم های عامل مختلف ». بنابراین نیاز به وجود یک API واحد که بتواند قدرتمند ، قابل انتقال و سطح پایین باشد به شدت احساس می شد. OpenGL این نیازها را به طور کامل برطرف کرده و خواهد کرد.

سیر تاریخی OpenGL

در این کتاب سعی بیشتر بر استفاده ی کاربردی از OpenGL است تا مباحث نظری و تاریخچه ی آن. اما آشنایی با سیر تکاملی OpenGL بسیار مفید خواهد بود.

OpenGL در دهه ۱۹۹۰ توسط شرکت Silicon Grapihcs متولد شد. اما کار استاندارد سازی و تکمیل آن توسط گروه ARB (OpenGL Architecture Review Board) انجام شد. این گروه متشکل از شرکت های بزرگی همچون مایکروسافت ، ، Intel ، ATI ، SGI ، 3DLabs ، Dell ، NVIDIA ، Sun ، IBM و غیره بودند که البته مایکروسافت جهت تولید محصول انحصاری خود (یعنی DirectX) از سال ۲۰۰۳ این گروه را ترک کرد. نسخه ی اول OpenGL توسط شرکت SGI در سال ۱۹۹۲ منتشر شد. نسخه ی دوم آن نیز توسط شرکت 3DLabs توسعه داده شد. هم اکنون (در زمان نگارش این کتاب) OpenGL در نسخه ی ۳.۱ به سر می برد که با توجه به توسعه ی سریع این سیستم پیش بینی می شود تا زمان انتشار کتاب به نسخه های بسیار جدیدتری برسد.

هدف اصلی توسعه ی OpenGL

OpenGL در اصل برای دو هدف اصلی توسعه داده شده و می شود :

۱- پنهان کردن پیچیدگی کار با سخت افزارهای مختلف گرافیکی با یک رابط سطح پایین

۲- ساده کردن کارهای گرافیکی خصوصا سه بعدی با ایجاد یک رابط استاندارد واحد

کتابخانه های مرتبط با OpenGL

همان طور که ذکر شد ، OpenGL مستقل از سکو است. بنابراین کارهایی که اختصاص به سکوی میزبان دارد را به طور مستقیم پشتیبانی نمی کند. از جمله ی این کارها می توان به ایجاد و کنترل پنجره ها ، کنترل ورودی و خروجی ، ایجاد اشیای آماده (مانند کره ، مخروط و ...) ، کنترل صداها و سیگنال های دیجیتال و ارتباط با شبکه را نام برد. دلیل پشتیبانی نکردن این موارد ، وابسته بودن آن ها به سیستم عامل و سکوی میزبان است که تفاوت قابل ملاحظه ای با هم دارند. بنابراین OpenGL این کارها را به کتابخانه های مرتبط واگذار کرده است. این کتابخانه ها قابلیت انجام تمامی موارد ذکر شده را دارند.

این کتابخانه ها بر دو نوع اند : ۱- کتابخانه های متسقل از سکو ۲- کتابخانه های وابسته به سکو

کتابخانه های مستقل از سکو همانند OpenGL قابلیت انتقال به هر سکویی که OpenGL به آن پورت شده باشد را دارند و دیگر نیازی به تغییر کدهایتان در آن ها ندارید. اما کتابخانه های وابسته به سکو در سیستم عامل های خاصی اجرا می شوند و در دیگر سیستم عامل ها کارایی ندارند. (نیاز به تغییر عمده ی کد وجود دارد)

از جمله ی کتابخانه های مستقل از سکو می توان دو کتابخانه ی GLUT و SDL را نام برد که در این کتاب بر روی GLUT مانور خواهیم داد.

از جمله ی کتابخانه های وابسته به سکو می توان به WGL برای ویندوز ، CGL برای Mac OS X و GLX برای لینوکس اشاره کرد. البته لازم به ذکر است این کتابخانه ها تنها استفاده از سیستم Windowing را فراهم می کنند و دیگر قابلیت ها مثل کنترل ورودی و خروجی یا صدا را پشتیبانی نمی کنند و برای اضافه کردن این قابلیت ها بایستی از زبان برنامه نویسی مورد استفاده و یا ویژگی های سیستم عامل خود استفاده کنید. پس ملاحظه می فرمایید که استفاده از کتابخانه های مستقل از سکو ضمن اینکه وابستگی به میزبان ندارند ، همگی موارد نیاز برای نوشتن یک بازی رایانه ای را برایتان فراهم می آورند.

کتابخانه ی SDL

SDL یک کتابخانه ی تمام عیار چند رسانه ای است (Simple DirectMedia Layer) که قابلیت انتقال به هر سکویی را داراست. شما در این کتابخانه با استفاده از زیر سیستم هایی که ایجاد شده اند ، قابلیت های زیر را خواهید یافت :

1- زیر سیستم SDL_image : قابلیت رندر کردن انواع فرمت های عکس

2- زیرسیستم SDL_mixer : قابلیت کنترل انواع صداها و مونتاژ کردن آن ها

3- زیرسیستم SDL_net : قابلیت انواع اتصالات به شبکه های کامپیوتری

4- زیرسیستم SDL_ttf و SDL_rtf : امکان رندر کردن فونت ها و قالب های فرمت rtf

همچنین این کتابخانه امکان کنترل Joystic را در اختیار شما قرار می دهد. با استفاده از سرآیند SDL_opengl.h شما می توانید به طور کامل از قابلیت های سطح پایین OpenGL در برنامه ی خود قرار دهید. متاسفانه به دلیل وسیع بودن بحث SDL در این کتاب به آن نمی پردازیم اما کارگاه های آنلاین بسیاری در وب وجود دارند که می توانید قدم به قدم این کتابخانه را یاد بگیرید. برای کسب اطلاعات بیشتر در این مورد به سایت رسمی این کتابخانه یعنی www.libsdl.org مراجعه فرمایید.

کتابخانه ی GLUT

در واقع جعبه ابزار OpenGL است. این کتابخانه تکمیل کننده ی کتابخانه ی GLU (OpenGL Utility Library) است که توابعی را ارائه می دهد که یک سطح بالاتر از سطح توابع اصلی OpenGL هستند. کتابخانه ی GLU معمولا به همراه OpenGL توزیع می شود چرا که کارهای اصلی که با

OpenGL می توان انجام داد از قبیل : تبدیلات مختصات ، ایجاد texture ها ، ایجاد چهارضلعی ها ، تولید خطاهای OpenGL و غیره را شامل می شود. و در واقع برنامه نویسی OpenGL بدون GLU کار بسیار سخت و طاقت فرسایی خواهد بود. حال کاری که کتابخانه ی GLUT کرده است (OpenGL Utility Toolkit) یک سطح بالاتر از GLU را ارائه داده است. یعنی کار با اجزای سیستم عامل میزبان. کارهایی از قبیل کنترل پنجره ها ، کنترل ورودی و ... را می توان با این کتابخانه انجام داد. سه کتابخانه ی GLU ، OpenGL و GLUT یک پکیج کامل و هماهنگ را جهت ایجاد برنامه های گرافیکی بسیار قدرتمند فراهم می سازند.

نوشتن اولین برنامه ی OpenGL در ویندوز

حال وارد قسمت عملی بحث شده و اولین برنامه ی خود را در سیستم عامل ویندوز می نویسیم. برنامه ها را در ویندوز می نویسیم اما به دلیل اینکه از GLUT برای نوشتن برنامه ها استفاده می کنیم ، قابلیت انتقال به هر سیستم عاملی را خواهند داشت. سیستمی که در طول تست برنامه ها از آن استفاده کرده ایم ، ویندوز XP به همراه Visual Studio 2005 است. البته شما می توانید از ویژیوال استودیو هم استفاده نکنید چرا که هیچ گونه کاری به قسمت Visual نداریم و فقط برنامه های Console می نویسیم. بنابراین می توانید از Borland C++ و یا GCC نیز در صورت تمایل استفاده کنید. همان طور که اطلاع دارید برنامه های OpenGL را تقریبا با تمامی زبان های برنامه سازی رایج می توانید بنویسید اما ما C++ را انتخاب کرده ایم ، به خاطر اینکه از تمامی قابلیت های OpenGL استفاده می کند و همچنین بیشتر دانشجویان کامپیوتر و کسانی که دروس کامپیوتری را خوانده اند با آن آشنا هستند و به صورت یک استاندارد زبان برنامه سازی است.

نصب کردن GLUT در ویندوز

شاید برایتان عجیب باشد (!) اما OpenGL به صورت پیشفرض در تمامی نسخه های ویندوز نصب است. (اگر باور ندارید به پوشه ی system32 در فولدر ویندوز خود بروید و فایل opengl32.dll را پیدا کنید.) همچنین GLU هم به صورت پیشفرض نصب است (باز هم پوشه ی system32 و فایل glu32.dll را ببینید) تنها کاری که نیاز دارید انجام دهید ، نصب GLUT است. ابتدا بایستی پکیج آن را از آدرس زیر دانلود کنید :

http://www.xmission.com/~nate/glut.html

بعد از دانلود آن را Extract کرده و بایستی فایل های زیر را در این پوشه داشته باشید :

glut32.dll glut32.dll glut.h

حال مراحل زیر را مرحله به مرحله انجام دهید :

1- فایل glut32.dll را به پوشه ی system32 از پوشه ی ویندوز خود کپی کنید.

2- فایل glut32.lib را به مسیر زیر کپی کنید (البته اگر استودیو را در C نصب کرده اید)

C:\Program Files\Visual Studio 8\VC\PlatformSDK\Lib\

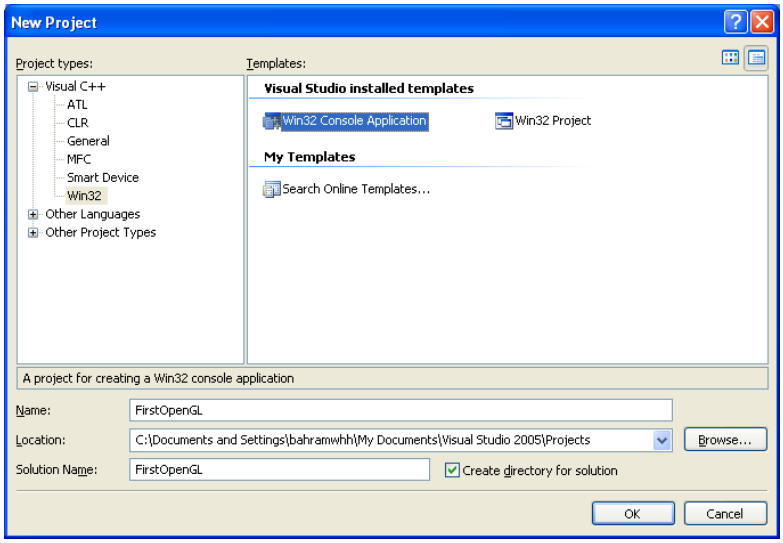
3- فایل glut32.h را به مسیر زیر کپی کنید :

C:\Program Files\Visual Studio 8\VC\PlatformSDK\Include\gl\

4- در انتها ویندوز خود را restart نمایید تا تغییرات انجام شود.

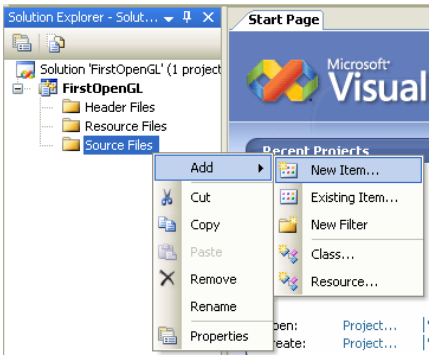
حال با انجام این مراحل بایستی GLUT در ویندوز شما نصب شده باشد. برای اطمینان از این موضوع یک برنامه ی ساده را با آن می نویسیم.

Visual Studio 2005 را باز کرده و از منوی File > New > Project را انتخاب کنید و در قسمت Project Types دکمه ی گسترش Visual C++ را بزنید و Win32را انتخاب کنید و از قسمت Visual Studio Installed Templates گزینه ی Win32 Console Application را بزنید و در قسمت Name یک نام انتخابی مثلا (FirstOpenGL) را بنویسید و دکمه ی OK را بزنید.



حال در پنجره ی ظاهر شده Next را بزنید. در پنجره ی بعدی دقت کنید که Empty Project در قسمت Additional options تیک خورده باشد و بر روی دکمه ی Finish کلیک کنید.

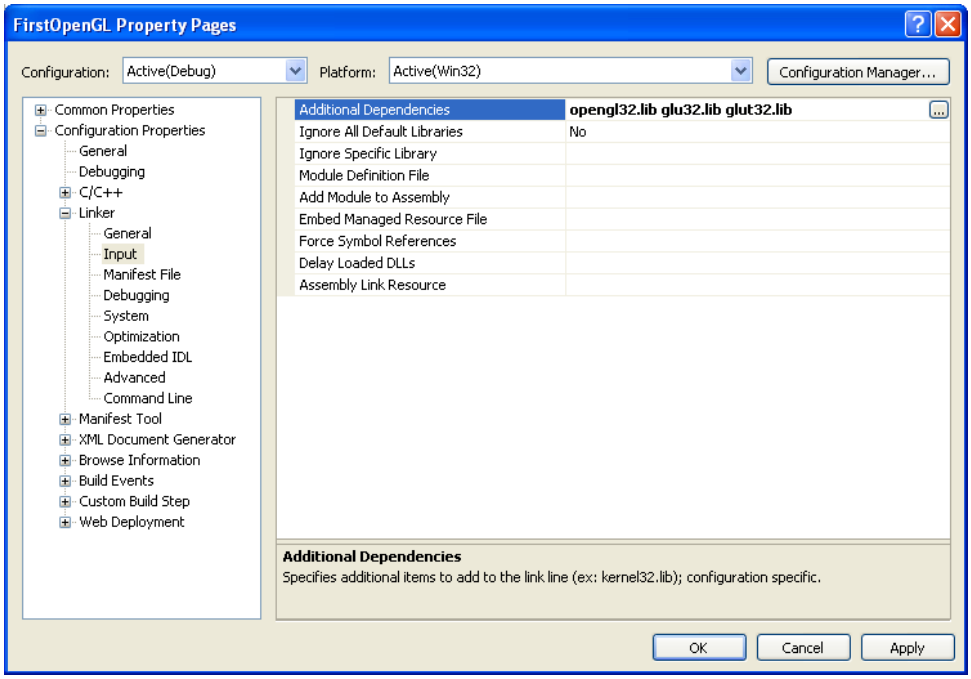
حال در قسمت Solution Explorer بر روی Source Files در پروژه ی خود کلیک راست کرده و از منوی Add گزینه ی New item را انتخاب کنید.



در این پنجره در قسمت Name نامی (مثل Simple) را وارد کرده و مطمئن شوید که File(.cpp C++) انتخاب شده باشد. و دکمه ی Add را بزنید. حال آماده اید که کد را بنویسید اما قبل از اینکه شروع به کد زدن بکنید بایستی OpenGL و GLUT را به کامپایلر بشناسانید. برای این کار از منوی Project در Visual Studio گزینه ی Properties FirstOpenGL را انتخاب کنید و در پنجره ی ظاهر شده علامت + قسمت Configuration Properties را بزنید تا گسترش یابد. سپس همین کار را در این قسمت برای Linker بکنید و فرزند Input از Linker را انتخاب کنید در قسمت سمت راست از این پنجره در قسمت Additional Dependencies عبارت زیر را وارد کنید:

opengl32.lib glu32.lib glut32.lib

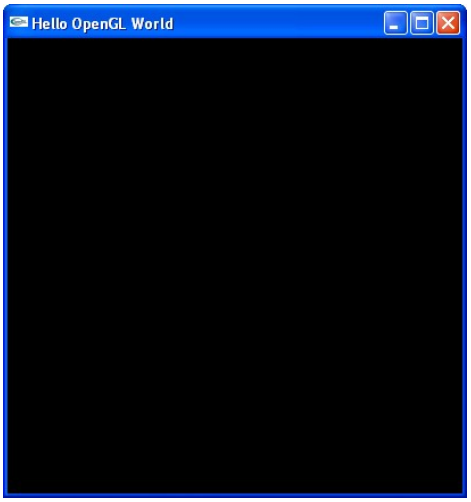
(بین هر کدام از libها یک فاصله بگذارید.) و دکمه ی OK را بزنید.



کار نصب و شناساندن GLUT به کامپایلر هم به طور کامل به اتمام رسیده است. حال برنامه ی بسیار ساده ی زیر را در قسمت code فایلی که هم اکنـــون در قسمت Source Files درست کردیم وارد کنید :

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

int Height=400, Width=400;

void display(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glutSwapBuffers();
}

int main(int argc, char **argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE);
	glutInitWindowSize(Width, Height);
	glutCreateWindow("Hello OpenGL World");

	glutDisplayFunc(display);
	glutMainLoop();
}
```

از منوی Build گزینه ی Build FirstOpenGL را بزنید (یا F7 را بزنید) و سپس (پس از کامپایل کردن بدون خطا ! اگـر خطـا گرفتیـد در یکـی از مراحل نصب اشتباه عمل کرده اید) در منوی Debug گزینه ی Start without debugging را بزنید. بایستی شکلی مشابه شـکل زیـر را بینیـد (یک صفحه ی سیاه با عنوان Hello OpenGL World) :



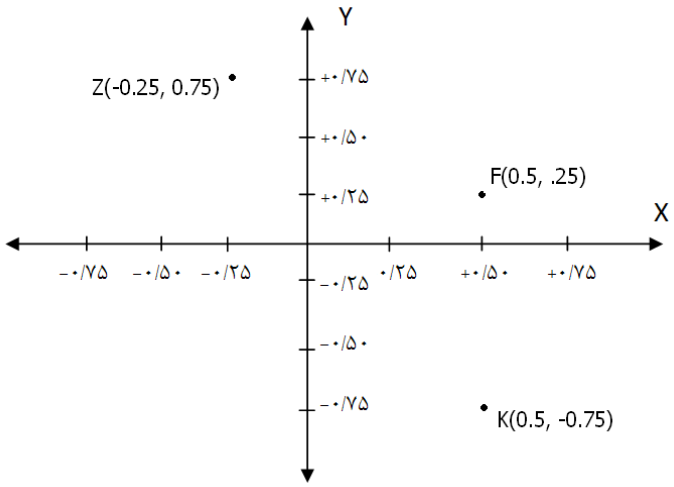
اگر این صفحه را دیدید به شما تبریک می گوییم شما هم اکنون اولین برنامه ی OpenGL خود را نوشته اید. (توضیح خطوط کد را در قسمت های بعدی این فصل خواهید دید.)

ورود به دنیای برنامه نویسی گرافیکی با OpenGL

حال می خواهیم کار اصلی خود را OpenGL انجام دهیم و رسما وارد دنیای برنامه نویسی گرافیکی شویم. برای این کار ابتدا از ۲ بعد برای برنامه نویسی استفاده می کنیم و سپس به ۳ بعد خواهیم پرداخت. در دنیای ۲ بعدی نحوه ی اختصاص مختصات و رسم اشیای ابتدایی را آموزش خواهیم داد. همچنین بعد از ۳ بعد به توضیح کامل توابع کتابخانه ی GLUT خواهیم پرداخت. فعلا نیازی به بحث در مورد کتابخانه ی glut نداریم ، فقط این نکته را در نظر داشته باشید که فرمی که در مثال اولیه (شروع کار) داشتیم را تا پایان کار خواهیم داشت و کدهای OpenGL خود را در تابع display خواهیم نوشت که در تابع main با استفاده از تابع glutDisplayFunc آن را به GLUT معرفی کرده ایم.

نحوه ی بیان مختصات در OpenGL

در OpenGL همه ی مختصات در بین دو عدد ۱- و ۱+ قرار می گیرند. مزیت بزرگ این کار قابلیت انواع تبدیلات برای این مختصات است. یعنی دیگر شما خود را از محدودیت های صفحه ی نمایش رها می سازید و تبدیلات واقعی در مقصد را OpenGL برایتان انجام می دهد. در دو بعد شما دو محور x و y را در اختیار دارید که هر کدام از ۱- تا ۱+ را برایتان فراهم می آورند که می توانید هر عدد بین این دو (شامل خود آن ها هم می شود) را انتخاب کنید. (مثلا ۰.۷) در فضای سه بعدی هم علاوه بر دو محور x و y ، محور z نیز که میزان فاصله ی آن از بیننده را تعیین می کند را در اختیار دارید. در شکل زیر مثالی در فضای دوبعدی را می بینید که سه نقطه را مشخص می کنند :



نحوه ی اختصاص رئوس

در OpenGL اکثر شکل ها با دادن رئوس به آن رسم می شوند. به عنوان مثال برای رسم یک مثلث کافی است که سه راس را به OpenGL بدهید و مثلث را برایتان رسم می کند. برای اعلان رئوس به OpenGL از تابع کلی glVertex استفاده می کنیم. (توجه کنید توابع پایه OpenGL با gl شروع می شوند. همانند توابع پایه glut که با glut شروع می شوند.) اما این تابع دارای چندین شکل مختلف است. که به صورت زیر است (دو حالت کلی)

۱- اگر نقاط را به صورت تک تک بدهید :

glVertex{234}(dfis)(مختصات)

۲- اگر نقاط را به صورت آرایه ای بدهید :

glVertex{234}(dfis)v(مختصات)

(به حرف v که ابتدای vector یا همان آرایه است توجه کنید)

قسمت {۲۳۴} به این معناست که قسمت (مختصات) می تواند ۲، ۳ و یا ۴ پارامتر بگیرد. که glVertex2 به معنای دو بعدی بودن مختصات داده شده و x و y را به عنوان دو پارامتر می گیرد. glVertex3 به معنای سه بعدی بودن مختصات داده شده و x و y و z را عنوان پارامتر خواهد گرفت. glVertex4 به معنای سه بعدی بودن مختصات داده شده و پارامتر چهارم میزان بزرگنمایی راس را مشخص می کند.

قسمت (dfis) نوع پارامترهایی که به این تابع را ارسال می کنید مشخص می کند. که مشخص کننده ی مقادیر زیر هستند :

- d -> double
- f -> float
- i -> integer
- s -> short

علاوه بر این ها نوع دوم مشخص کردن رئوس به صورت آرایه ای است. یعنی ابتدا رئوس ۲ گانه یا سه گانه را در یک آرایه نسبت می دهید و سپس آن را به تابع ارسال می کنید. چند مثال از هر دو نوع را در زیر می بینید :

```
glVertex2i(2, 4);
```

راسی را در نقطه ی ۲, ۴ از صفحه ی مختصات دو بعدی مشخص می کند.

```
GLfloat v[3] = {2.0, -9.0, +4.0};
glVertex3fv(v);
```

ابتدا یک آرایه با سه مولفه از نوع GLfloat تعریف کرده و سپس تنها آن آرایه را به تابع تحویل می دهیم. (توجه کنید که نقاط داده شده اگرچه بین ۱- و ۱+ نیستند ولی به طور خودکار به این دو عدد تبدیل می شوند)

نحوه ی اختصاص رنگ ها

شما در دنیای واقعی وقتی می خواهید که چیزی بکشید و یا حتی چیزی بنویسید ابتدا رنگ مناسب خود را انتخاب می کنید. OpenGL هم برای رسم رئوس ابتدا بایستی رنگ آن ها را بداند. به طور کلی از تابع عمومی glColor استفاده می شود که شکل کلی آن به صورت زیر است :

```
glColor{34}(bdfisubusui)v(رنگ مورد نظر)
```

همانند تابع glVertex این تابع می تواند ۳ و یا ۴ پارامتر بپذیرد. نوع ۳ پارامتری رنگ های RGB (قرمز ، سبز و آبی) را مشخص می کنند. و پارامتر چهارم نیز میزان شفافیت رنگ (Alpha) را مشخص می کند. نوع داده هایی که این پارامترها می توانند باشند به صورت زیر است :

- b (byte)
- d (double)
- f (float)
- i (integer)
- s (short)
- ub (unsigned byte)
- us (unsigned short)
- ui (unsigned integer)

همچنین به این تابع می توانید به جای مشخص کردن تک تک پارامترها یک آرایه نسبت دهید (دلخواه) . در زیر چند مثال از این تابع را می بینید :

```
glColor4f(1.0, 0.5, 1.0, 0.2);
```

رنگی مایل به زرد با میزان شفافیت ۰.۲ ایجاد خواهد شد.

```
GLint c[3] = { 255, 0, 0 };
glColor3i(c);
```

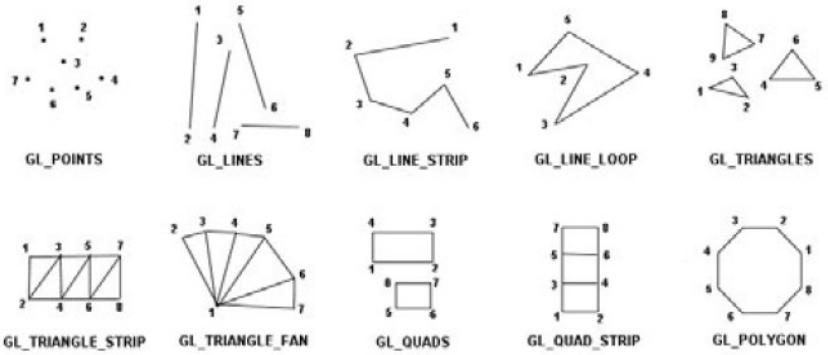
رنگ قرمز (استفاده از نوع آرایه ای اختصاص رنگ)

بلاک های دستور OpenGL

وقتی می خواهید شکلی را ایجاد کنید بایستی آن را در بلاک های دستور OpenGL که با glBegin() و glEnd() مشخص مـی شـوند قـرار دهیـد. روش کلی بدین صورت است که ابتدا رنگ شکل مورد نظر را خارج از این بلاک تعیین می کنید و سپس با استفاده از glBeign() و ارسال پارامتر مربوطه نوع شکلی که قرار است OpenGL برایتان رسم کند را مشخص می کنید و سپس رئوس شکل را در داخل این بلاک می دهید (بسته به نوع شکل تفاوت می کند) و در انتها بلاک را با glEnd() می بندید. (در ادامه با حل مثال هایی بیشتر با این بلاک ها آشنا می شوید)

رسم اشکال ساده و ۲ بعدی در OpenGL :

اشکال دو بعدی که در OpenGL می توانید رسم کنید عبارتند از : نقطه ، خط ، مثلث ، ۴ ضلعی و چند ضلعی (کـه البتـه هـر کـدام از ایـن هـا دارای چندین نوع پیوسته و گسسته می باشند) که در شکل زیر خود اشکال و همچنین نحوه ی رسم آن ها را ملاحظه می فرمایید :



در ادامه نحوه ی رسم هر کدام از این اشکل را با مثال هایی مبسوط توضیح خواهیم داد. جهت جلوگیری از تکرار کـدهای ثـابت فقـط کـدهای داخـل تـابع diplay را خواهیم نوشت.

رسم نقطه (GL_POINTS) :

قبل از هر چیز بایستی بتوانید یک نقطه را در صفحه رسم کنید. برای رسم یک نقطه در مختصات (۰.۲, -۰.۵) تابع display را به صورت زیر تغییر مـی دهیم (خطوط زیر را در بین دو خط موجود در تابع display قرار دهید) :

```
glColor3f(1.0, 0.0, 1.0);
glBegin(GL_POINTS);
glVertex2f(0.2, -0.5);
glEnd();
```

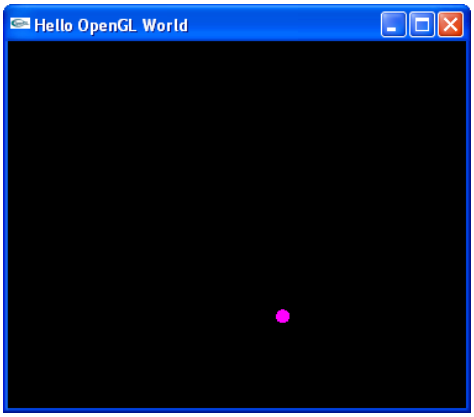
نتیجه ای که خواهید گرفت ، یک نقطه ی کوچک صورتی رنگ در ربع چهارم صفحه ی مختصات است. اما اگر ملاحظه کنید این نقطه بسیار کوچـک اسـت شما می توانید با استفاده از تابع glPointSize اندازه ی نقاط خـود را تغییـر دهیـد (تـا ۶۴ مـی توانیـد بـزرگ کنیـد) . بنـابراین خـط زیـر را قبـل از glBegin اضافه کنید :

```
glPointSize(20.0);
```

در صورتی که برنامه ی خود را اجرا کنید به جای نقطه ، یک مربع خواهید گرفت. دلیل این امر هم مربعی بودن شکل کلی پیکسل ها است. برای اینکه نقاط بزرگ شده ی خود را به صورت دایره ای ببینید بایستی GL_POINT_SMOOTH را فعال کنید. برای فعال کردن ویژگی ها از قبیل همین ویژگی بایستی از تابع glEnable در OpenGL استفاده کنید (که بعدا بیشتر در مورد آن بحث خواهیم کرد) بنابراین خطوط زیر را در تابع display قرار می دهیم :

```
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 0.0, 1.0);
glEnable(GL_POINT_SMOOTH);
glPointSize(20.0);
glBegin(GL_POINTS);
glVertex2f(0.2, -0.5);
glEnd();
glutSwapBuffers();
```

نتیجه به صورت شکل زیر است :



در این قسمت تنها هدف اصلی رسم یک نقطه بود و توابع اضافی که گفته شد صرفا جهت آشنایی بیشتر با این نقاط بود. در مراحل بعدی مثال های پیشرفته تری از نقاط را انجام خواهیم داد.

نکته : شما می توانید برای رسم چندین نقطه از چندین بلاک glBegin و glEnd استفاده کنید. برای مثال کد زیر دو نقطه را رسم می کند :

```
glBegin(GL_POINTS);
glVertex2f(0.2, -0.5);
glEnd();

glBegin(GL_POINTS);
glVertex2f(0.8, 0.7);
glEnd();
```

اما این کار تنها وقت شما را تلف می کند و بر خوانایی برنامه ی شما اثر می گذارد. این دو نقطه (و یا تعداد بیشتر) را به راحتی می توانید در یک بلاک شروع و پایان جای دهید. به صورت زیر :

```
glBegin(GL_POINTS);
glVertex2f(0.2, -0.5);
glVertex2f(0.8, 0.7);
glEnd();
```

رسم خطوط (GL_LINES) :

برای رسم خطوط سه انتخاب دارید :

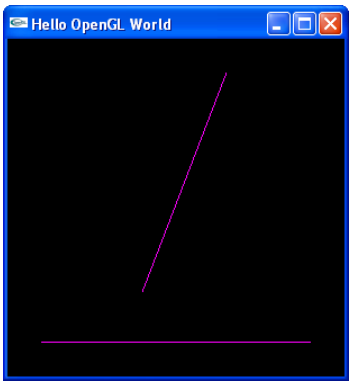
- رسم پاره خط های جدا ازهم (GL_LINES)
- رسم خطوط به هم پیوسته با دو سر انتهایی باز (GL_LINE_STRIP)
- رسم خطوط به هم پیوسته با دو سر انتهایی به هم بسته (GL_LINE_LOOP)

که در ادامه از هر کدام مثالی خواهیم آورد:

رسم پاره خط های جدا از هم :

برای این کار کافی است که دو راس (ابتدایی و انتهایی) را در بلوک شروع و پایان OpenGL قرار دهید و GL_LINES را به عنوان پارامتر بـه glBegin بفرستید. جهت اینکه بیش از یک خط رسم کنید کافی است که رئوس را به صورت دوتایی در بلاک شروع و پایان قرار دهید تا به تعـداد آن هـا خـط رسـم شـده دریافت کنید. مثال زیر دو خط را برای شما به صورت زیر رسم می کند :

```
glBegin(GL_LINES);
glVertex2f(-0.2, -0.5);
glVertex2f(0.3, 0.8);

glVertex2f(-0.8, -0.8);
glVertex2f(0.8, -0.8);
glEnd();
```

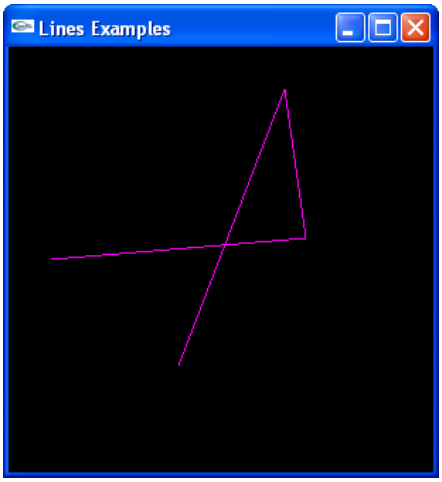


رسم خطوط پیوسته با دو سر انتهایی باز :

تفاوت اصلی این نوع خطوط این است که هنگامی که بیش از دو راس را در بلوک آغاز و پایان قرار می دهید راس آغازین بعدی را به آخریـن راس پایـانی متصـل می کند. در واقع بعد از مشخص کردن دو راس اول دیگر نیازی به ارائه ی دو راس باهم نیست، چرا که راس آغازین همـان راس پایـانی خـط قبـل اسـت. در زیـر مثالی از سه خط به هم پیوسته را می بینید :

```
glBegin(GL_LINE_STRIP);
glVertex2f(-0.2, -0.5);
glVertex2f(0.3, 0.8);
```

```
glVertex2f(0.4, 0.1);

glVertex2f(-0.8, 0.0);
glEnd();
```

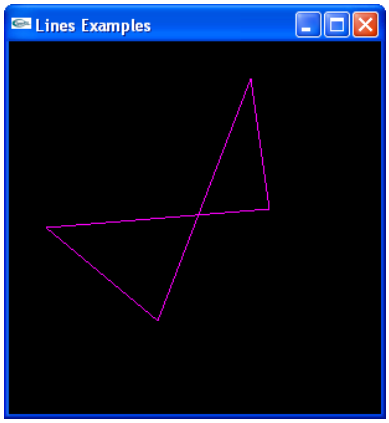


رسم خطوط پیوسته با دو سر انتهایی به هم بسته :

همان طور که از نام گذاری که کرده ایم معلوم است ، تفاوت آن با نوع قبلی به هم متصل شدن راس اول و راس آخر در هنگام مشاهده ی تصویر است :

و بایستی به تابع glBegin پارامتر GL_LINE_LOOP را فرستاد. مثال قبلی را با همین تفاوت مشاهده می کنید :

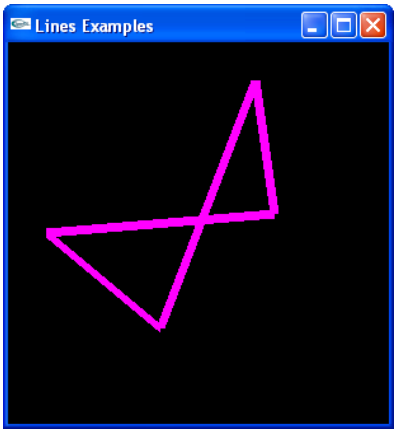
```
glBegin(GL_LINE_LOOP);
glVertex2f(-0.2, -0.5);
glVertex2f(0.3, 0.8);

glVertex2f(0.4, 0.1);

glVertex2f(-0.8, 0.0);
glEnd();
```



نحوه ی تغییر عرض خطوط :

با استفاده از تابع glLineWidth() و دادن یک عدد اعشاری می توان اندازه ی عرض خط را تغییر داد. مثال قبلی را با استفاده از این تابع بزرگنمایی می کنیم (تنها خط زیر را قبل از glBegin قرار می دهیم) :

glLineWidth(7.0);



مباحث پیشرفته تری از خطوط را در قسمت های بعدی خواهید دید.

رسم مثلث :

برای رسم مثلث ها نیز OpenGL سه گزینه را در اختیار شما قرار می دهد. اما قبل از بررسی آن ها به این نکته توجه کنید که تمامی اشکالی که OpenGL برای ما رسم می کند ، به صورت پاد ساعتگرد رسم خواهد کرد (با حل مثال هایی با این موضوع بیشتر آشنا خواهید شد).

انواع مثلث های موجود :

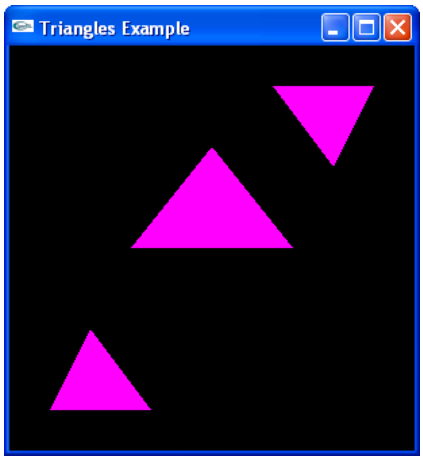
- مثلث های جدا از هم
- مثلث های دوبدو مشترک در یک ضلع
- مثلث های مشترک در یک راس واحد

مثلث های جدا از هم :

برای رسم هر کدام از این مثلث ها بایستی سه راس را مشخص کنید. برای مثال برای رسم سه مثلث بایستی ۹ راس را مشخص کنید. این مثلث ها به خودی خود جدا از هم هستند مگر اینکه رئوس را طوری بدهید (به صورت دستی) که اشتراکی بین آن ها ایجاد کنید. ولی برای مثلث هایی که با هم اشتراک دارند ، انواع بعدی توصیه می شود. به مثالی در این زمینه که سه مثلث جدا از هم را رسم می کند توجه کنید :

```
glBegin(GL_TRIANGLES);
glVertex2f(-0.8, -0.8);
glVertex2f(-0.6, -0.4);
glVertex2f(-0.3, -0.8);

glVertex2f(0.8, 0.8);
glVertex2f(0.6, 0.4);
glVertex2f(0.3, 0.8);

glVertex2f(-0.4, 0.0);
glVertex2f(0.0, 0.5);
```

```
glVertex2f(0.4, 0.0);

glEnd();
```

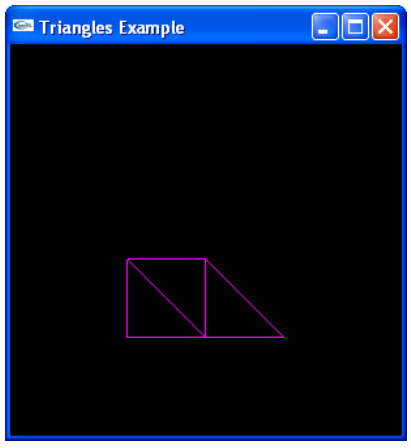


مثلث های دوبدو مشترک در یک ضلع :

این نوع مثلث ها برای رسم مثلث های پیوسته بسیار در وقت و کد صرفه جویی می کنند. نحوه ی رسم آن ها بدین صورت است که ابتدا بایستی سه راس مثلث اول را رسم کنید و از آن به بعد تنها با دادن یک راس ، یک مثلث بر روی ضلعی که به آن نقطه نزدیک تر است ، ساخته می شود. جهت رسم این نوع مثلث ها بایستی به glBegin پارامتر GL_TRIANGLE_STRIP را بفرستید. به مثالی در این زمینه توجه کنید :

توجه : به دلیل اینکه مثلث های جدا از هم تشخیص داده شوند از تابع glPolygonMode استفاده کرده ایم (مثلث ها از نوع چند ضلعی هستند که در قسمت بعد توضیح داده می شود) تا نقاط به طور کامل مشخص شوند.

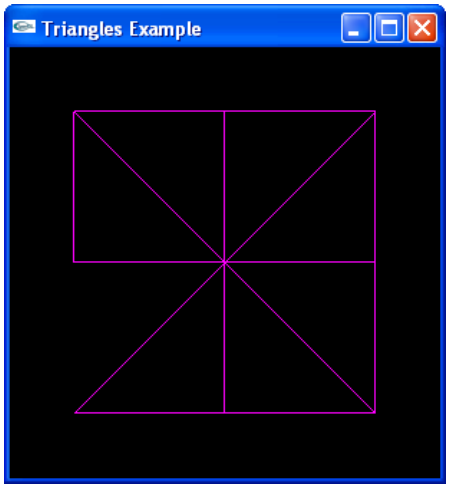
```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLE_STRIP);
glVertex2f(-0.4, -0.5);
glVertex2f(-0.4, -0.1);
glVertex2f(0.0, -0.5);

glVertex2f(0.0, -0.1);

glVertex2f(0.4, -0.5);
glEnd();
```



مثلث های مشترک در یک راس واحد :

مشابه مثلث های قبلی هستند با این تفاوت که یک راس بین تمامی مثلث ها مشترک است. (و آن راس همان راس اولی است که در بلاک شروع و پایان تعریف می کنید) در مثال زیر ۷ مثلث را با استفاده از ۹ راس رسم کرده ایم :

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLE_FAN);

glVertex2f(-0.0, 0.0);
glVertex2f(-0.7, 0.0);
glVertex2f(-0.7, 0.7);

glVertex2f(0.0, 0.7);
glVertex2f(0.7, 0.7);
glVertex2f(0.7, 0.0);
glVertex2f(0.7, -0.7);
glVertex2f(0.0, -0.7);
glVertex2f(-0.7, -0.7);

glEnd();
```



رسم چند ضلعی (GL_POLYGON) :

با استفاده از ویژگی چند ضلعی ها می توانید انواع چند ضلعی ها را رسم کنید. همان طور که در بحث رسم مثلث ها هم گفته شد ، مثلث ها نیز از چند ضلعی ها مشتق شده اند. اما به دلیل اینکه مثلث ها دارای تنوع سه گانه ی جالبی هستند و همواره نیز محدب هستند و رئوس یک مثلث نمی توانند یکدیگر را قطع کنند ، برای رسم مثلث هیچگاه از چند ضلعی استفاده نمی شود. بنابراین برای رسم چند ضلعی های دارای بیش از ۴ ضلع از این نحوه ی رسم استفاده می شود. چند ضلعی ها به طور پیشفرض توپر رسم می شوند اما شما با استفاده از تابع glPolygonMode می توانید آن را تغییر دهید. البته بیشترین کاربرد این تابع در ۳ بعد است که در قسمت های بعدی به طور کامل به آن خواهیم پرداخت. ولی شکل کلی به کار گیری این تابع به صورت زیر است :

```
void glPolygonMode(GLenum face, GLenum mode);
```

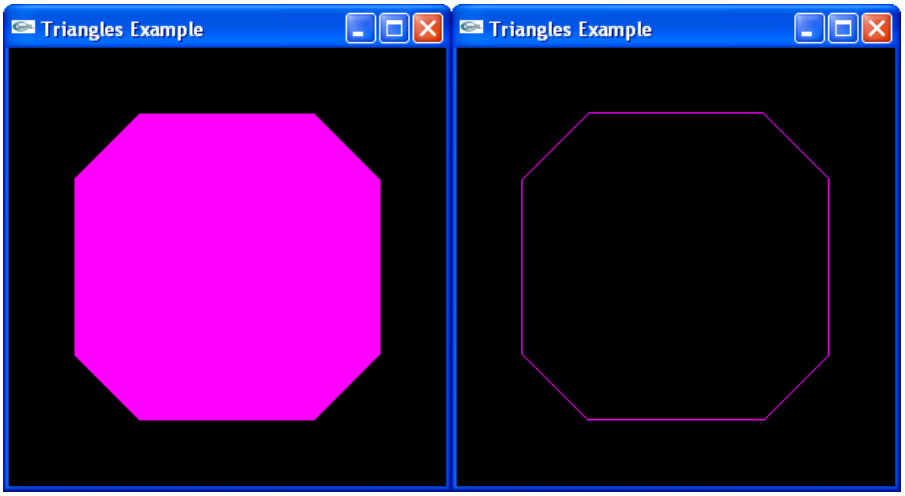
آرگومانی که به پارامتر face ارسال می کنید می تواند یکی از مقادیر GL_BACK ، GL_FRONT یا GL_FRONT_AND_BACK باشد. این پارامتر مشخص کننده ی نحوه ی تاثیر پارامتر mode بر چندضلعی است. هنگامی که به بحث های دریچه ی دید و تغییر زاویه ی دوربین برسیم ، ملاحظه می کنید که تمامی اشکال دارای سه قسمت جلو ، کناره و پشت هستند که می توانید آن ها را ببینید. و هر کدام از این قسمت ها می توانند توپر و یا خالی باشند. پارامتر mode می تواند یکی از سه مقدار GL_LINE ، GL_POINT و یا GL_FILL را بگیرد (که GL_FILL به صورت پیشفرض انتخاب شده

است) GL_POINT فقط باعث می شود که نقاط ابتدا و انتهایی خطوط نمایش داده شوند. GL_LINE باعث نمایش تنها خطوط بجای نمایش توپر می شود و در نهایت GL_FILL که باعث نمایش توپر شکل به صورت پیشفرض می شود.

این تابع بر روی مثلث و مربع ها تاثیر گذار است و می توانید به طور کامل بر روی آن ها نیز اعمال کنید. به این بحث در قسمت های پیشرفته تر باز خواهیم گشت.

در مثال زیر یک ۸ ضلعی را به صورت توپر و غیر توپر می بینید (کد به صورت غیر توپر نوشته شده است) :

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);

glVertex2f(-0.7, -0.4);
glVertex2f(-0.7, 0.4);

glVertex2f(-0.4, 0.7);
glVertex2f(0.4, 0.7);

glVertex2f(0.7, 0.4);
glVertex2f(0.7, -0.4);

glVertex2f(0.4, -0.7);
glVertex2f(-0.4, -0.7);

glEnd();
```



رسم چهارضلعی :

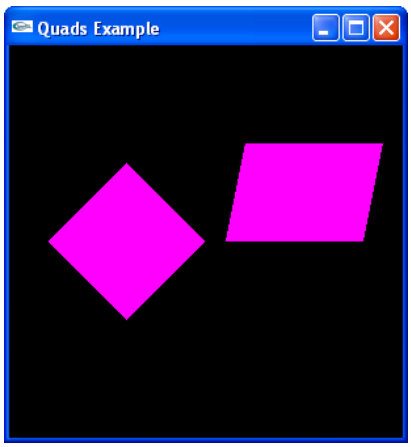
این نوع از اشکال نیز از چند ضلعی ها مشتق شده اند و در واقع هر چهارضلعی با استفاده از توابع رسم چند ضلعی انجام می گیرد. برای رسم هر چهارضلعی دو گزینه پیش رو دارید :

- چهارضلعی های جدا از هم (GL_QUADS)
- چهارضلعی های مشترک در یک ضلع (GL_QUAD_STRIP)

برای رسم چهارضلعی های جدا از هم همانند رویه ی قبلی عمل می کنیم. برای هر چهارضلعی چهار راس جداگانه در نظر می گیریم تا OpenGL آنها را به هم متصل نموده و یک مستطیل، مربع، لوزی و یا هر چهارضلعی دیگری را برایمان رسم کند.

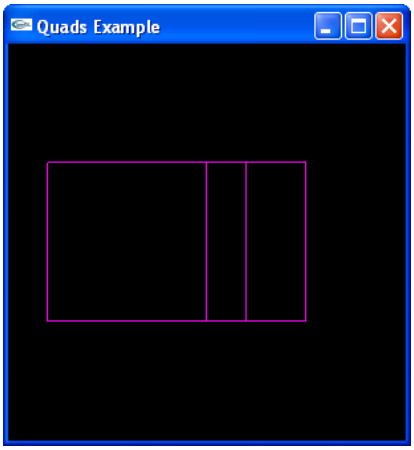
در مثال زیر یک لوزی و یک متوازی الاضلاع را با استفاده از چهارضلعی های جدا از هم کشیده ایم :

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glBegin(GL_QUADS);

glVertex2f(-0.8, 0);
glVertex2f(-0.4, 0.4);
glVertex2f(0.0, 0.0);
glVertex2f(-0.4, -0.4);

glVertex2f(0.9, 0.5);
glVertex2f(0.2, 0.5);
glVertex2f(0.1, 0.0);
glVertex2f(0.8, 0.0);

glEnd();
```



برای رسم چند ضلعی های مشترک در یک ضلع کافی است که ابتدا چهار راس اولیه برای چهارضلعی اصلی را داده و سپس برای دیگر اشکال تنها ۲ راس را مشخص می کنیم. دقت کنید که چهار ضلعی بعدی بر روی آخرین ضلع چهارضلعی قبلی که رسم کرده اید ، رسم می شود. برای رسم این گونه اشکال بایستی GL_QUAD_STRIP را به glBegin ارسال کنیم. ترتیب نوشتن رئوس در این نوع اشکال بسیار مهم است. به شکل ابتدایی اشکال ۲ بعدی که در این بخش داده شد، دوباره توجه کنید و قسمت شماره ها را به خاطر بسپارید و سپس به مثال زیر که سه مستطیل به هم پیوسته را رسم می کند ، توجه کنید :

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_QUAD_STRIP);

glVertex2f(-0.8, 0.4);
glVertex2f(-0.8, -0.4);
glVertex2f(0.0, 0.4);
glVertex2f(0.0, -0.4);

glVertex2f(0.2, 0.4);
glVertex2f(0.2, -0.4);

glVertex2f(0.5, 0.4);
glVertex2f(0.5, -0.4);

glEnd();
```



تا کنون شکل های پایه در دو بعد در OpenGL را بررسی کردیم. اما اکنون این پرسش مطرح می شود که آیا واقعا تمام چیزهایی که ما نیاز داریم با استفاده از این اشکال پایه قابل رسم و پیاده سازی هستند؟! به طور مثال ما چیزی به نام دایره (با وجود کاربرد زیاد در تمامی گرافیک های واقعی) در اشکال پیدا نکردیم !

جواب بله است. هنگامی که شما می توانید خط (و نقطه) رسم کنید در واقع هر چیزی را می توانید رسم کنید. چرا که تمامی اشکال دیگر نیز (مانند مثلث ، مربع و یا چندضلعی) نیز با استفاده از همین خط و نقطه تشکیل شده اند و تنها برای صرفه جویی در زمان ، کد نوشته شده و همچنین دقت بیشتر در کار این اشکال پایه فراهم شده اند. برای کشیدن اشکال دیگر تنها مقداری ابتکار و دانش (در زمینه ی هندسه) نیاز است.

اکنون مثال کاملی را جهت تفهیم و تثبیت مطالب ارائه شده در این قسمت ، بررسی می کنیم.

مثال : دایره ای توپر به شعاع ۹/۰ که با رنگ آبی پر شده است رسم کنید ولی مربعی در مرکز آن به ضلع ٤/۰ رسم کنید که با رنگ زرد پر شده باشد.

همان طور که می دانید ، شکل پایه ای به نام دایره در اشکال پایه ی OpenGL وجود ندارد. اما نگران نباشید (!) به راحتی با استفاده از تعریف دایره می توانیم آن را بکشیم. اگر با تعریف مکان هندسی در هندسه آشنا باشید ، تعریف زیر را خواهیم داشت :

« دایره مکان هندسی نقاطی است که فاصله ی آنها از یک نقطه ی ثابت (مرکز) برابر مقدار ثابتی (شعاع) است. »

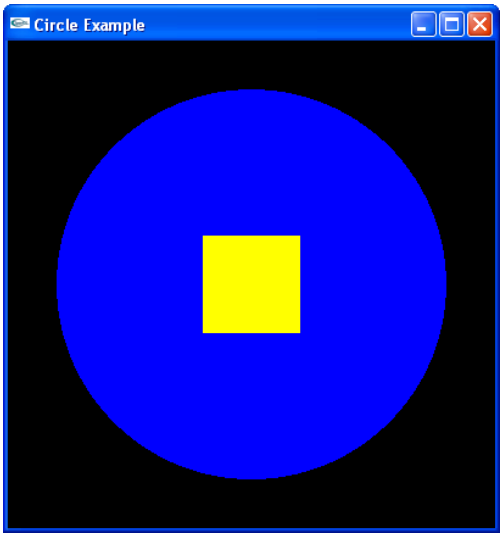
برای اینکه این مکان های ثابت را بدست آوریم یک فرمول ثابت داریم. باز هم در هندسه داریم که محیط دایره برابر است با : $2\pi r$

اما برای کشیدن دایره می توانیم دو شکل پایه استفاده کنیم : ۱- خطوط به هم پیوسته ۲- چندضلعی ها

در هر کدام از خطوط به هم پیوسته و چندضلعی ها اگر تعداد رئوس زیاد شود (و رئوس در یک مسیر دایره ای داده شوند) شکل مورد نظر ما به سمت دایره میل خواهد کرد. حال با توجه به این توضیحات کد کامل برنامه ی نوشته شده را در زیر آورده و سپس به توضیح خطوط کد می پردازیم :

```
1  #include <GL/glut.h>
2  #include <math.h>

3  int Height=400, Width=400;

4  #define edgeOnly 0

5  void DrawCircle(double radius, int numberOfSides)
6  {
```

```
7    // if edge only, use line strips; otherwise , use polygons
8    if(edgeOnly)
9      glBegin(GL_LINE_STRIP);
10   else
11     glBegin(GL_POLYGON);

12   // calculate each vertex on the circle
13   for ( int vertex = 0; vertex < numberOfSides; vertex++ )
14   {

15      // calculate the angle of current vertex
16      // ( vertex # * 2 * PI ) / # of sides
17      float angle_c = (float) vertex * 2.0 * 3.14159 / numberOfSides;

18      glColor3f(0,0,1); // Blue Color

19       // draw the current vertex at the correct radius
20      glVertex2f(cosf(angle_c)*radius, sinf(angle_c)*radius);
21 }

22   // if drawing edge only, then need to complete the loop with first vertex
23   if(edgeOnly)
24      glVertex2f(radius, 0.0);

25   glEnd();

26 }

27 void display(void)
28 {
29    glClear(GL_COLOR_BUFFER_BIT);

30    DrawCircle(0.8, 100);

31    glColor3f(1, 1, 0);
32    glBegin(GL_QUADS);
33    glVertex2f(-0.2, 0.2);
34    glVertex2f(-0.2, -0.2);
35    glVertex2f(0.2, -0.2);
36    glVertex2f(0.2, 0.2);
37    glEnd();

38    glutSwapBuffers();
39 }

40 int main(int argc, char **argv)
41 {
42    glutInit(&argc, argv);
43    glutInitDisplayMode(GLUT_DOUBLE);
44    glutInitWindowSize(Width, Height);
45    glutCreateWindow("Circle Example");

46    glutDisplayFunc(display);
47
48    glutMainLoop();
49 }
```



توضیح کد :

از همان شکل اولیه ای که تا کنون استفاده کردیم ، در اینجا نیز استفاده شده است. تنها چیزهایی که اضافه شده است تابع DrawCircle و رسم یک مربع در تابع display است. در خط ۲ سرآیند math را اضافه کرده ایم چرا که در تابع رسم دایره به آن نیاز داریم. خط ۳ تکراری است و طول و عرض پنجره ی OpenGL را مشخص می کند. در خط ۴ ثابت edgeOnly را تعریف کرده ایم که اگر برابر یک مقدار غیر صفر قرار دهید ، از چند ضلعی و اگر برابر صفر قرار دهید ، از خطوط به هم پیوسته برای رسم دایره استفاده می شود. که در حالت اول دایره ی رسم شده توپر ، و در حالت دوم دایره توخالی خواهد بود. در خط ۵ تابع DrawCircle دو آرگومان دریافت می کند که radius شعاع دایره و `numberOfSides` تعداد ضلع هایی که دایره را بوجود می آورند می باشد که همان طور که توضیح داده شد ، هر چقدر این پارامتر مقدار بیشتری بگیرد ، شکل نتیجه شده به دایره نزدیک تر خواهد بود. در خطوط ۷ تا ۱۱ با توجه به ثابت edgeOnly شرطی برقرار می شود تا از GL_POLYGON استفاده شود یا از GL_LINE_STRIP . در خط ۱۳ حلقه ای ایجاد می شود تا متغیر vertex را به تعداد ضلع هایی که به عنوان آرگومان ارسال کرده اید ، شمارش کند (و تکرار کند). در خط ۱۷ با استفاده از فرمول داده شده ، زاویه ی هرکدام از ضلع ها بدست می آید و در خط ۲۰ با استفاده از ضرب کردن کسینوس (ضلع مجاور) و سینوس (ضلع مقابل) این زاویه در شعاع دایره ، مکان دقیق هرکدام از رئوس پیدا شده و آن ها را به عنوان آرگومان به تابع دوبعدی glVertex2f می فرستیم. در خط ۱۸ نیز رنگ مورد نظر برای اضلاع تعیین می شود. در نهایت در خط ۲۳ تصمیم گیری می شود در صورتی که از خطوط به هم پیوسته استفاده شده است ، ابتدا و انتهای دو سر خطوط به هم متصل شود. در تابع diplay نیز پس از فراخوانی تابع DrawCircle با دو آرگومان ۸/۰ برای شعاع و ۱۰۰ برای تعداد اضلاع ، یک مربع به طول اضلاع ۴/۰ با رنگ زرد (در خط ۳۱) رسم شده است. بقیه ی خطوط نیز استفاده از تابع glut را نشان می دهند که همانند مثال های قبلی است.

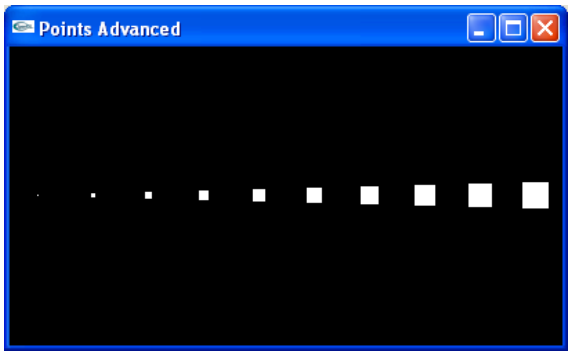
مباحث پیشرفته تر در مورد خط و نقطه :

در صورتی که تا اینجا تمامی مثال ها را اجرا و تست کرده باشید ، تقریبا با برنامه نویسی OpenGL راه افتاده اید. مباحثی که در این بخش مطرح می شوند را از مطالب قبلی جدا کردیم تا مقداری آشنایی با کلیت بحث پیدا کرده و سپس وارد مباحث تکمیلی بشوید. اکنون به ویژگی های تکمیلی در خط و نقطه که در مورد آن ها بحث کردیم ، می پردازیم :

تغییر اندازه ی نقاط به صورت پویا (یک مثال)

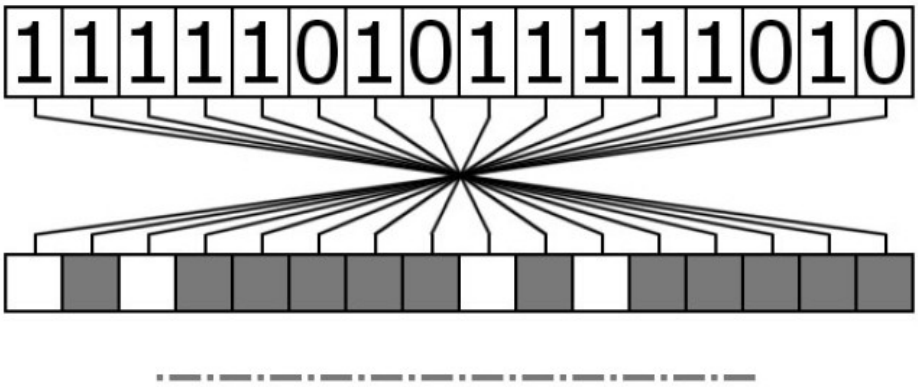
در قسمت رسم نقاط نحوه ی تغییر اندازه ی نقاط را با استفاده از glPointSize توضیح دادیم. حال مثال زیر را بررسی می کنیم که به صورت پویا نقاطی را در صفحه ی OpenGL بزرگ می کنیم. کد زیر را در قسمت تابع display از برنامه ی پایه (اولیه) جاگذاری کنید (کد ۰۳.cpp) :

```
glClear(GL_COLOR_BUFFER_BIT);

float Size = 1.0;
for(float pos=-0.9; pos < 0.9 ; pos+=0.2)
{
	glPointSize(Size);

	glBegin(GL_POINTS);
	glVertex2f(pos, 0.0);
	glEnd();

	Size += 2.0;
}
glutSwapBuffers();
```



ایجاد یک الگو برای خطوط

تا کنون ، خطوطی که رسم کردیم تنها خطوطی بودند که از ابتدای خط تا انتها بهم پیوسته و هیچگونه بریدگی نداشت. حال به طور مثال بخواهید با دادن مختصات یک خط ، یک نقطه چین داشته باشید ، چکار بایستی بکنید. این کار از طریق ایجاد الگو برای خط امکان پذیر است. در ابتدا شکل زیر را برای تفهیم بیشتر الگو ملاحظه کنید :

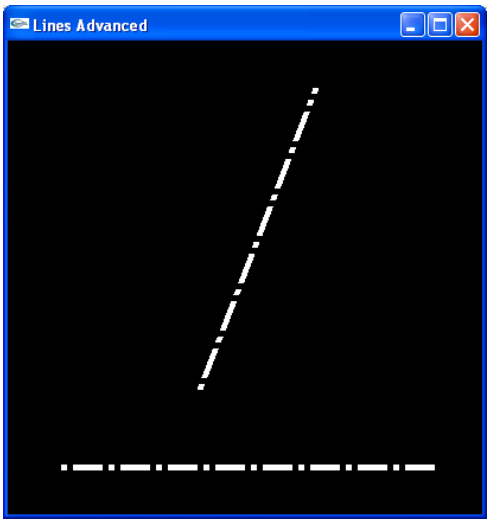


همان طور که می بینید الگوی خطوط یک عدد ۱۶ بیتی باینری است که اعمال الگو از راست به چپ برای خطوط انجام می شود. این الگو را بایستی در یک متغیر GLushort ذخیره کنید (و یا مستقیما به تابع بفرستید) . تابع اعمال الگوی خطوط به صورت زیر است :

void glLineStipple(GLint factor, GLushort pattern);

pattern همان الگوی ۱۶ بیتی است که بایستی به تابع داده شود و factor که از نوع integer است میزان تکرار بیت های ۱ را نشان می دهد. به عنوان مثال در صورتی که factor برابر ۵ باشد بعد از مواجه شدن با هر بیت ۱ به اندازه ی ۵ واحد خط تکرار می گردد. البته قبل از استفاده از این تابع بایستی GL_LINE_STIPPLE را با استفاده از glEnable فعال کرده باشید. در مثال زیر دو خط را با فاکتور ۵ رسم کرده ایم :

```
glClear(GL_COLOR_BUFFER_BIT);

glEnable(GL_LINE_STIPPLE);
GLushort stPattern = 0xFAFA;
glLineStipple(5, stPattern);

glLineWidth(5.0);
glBegin(GL_LINES);
glVertex2f(-0.2, -0.5);
glVertex2f(0.3, 0.8);

glVertex2f(-0.8, -0.8);
glVertex2f(0.8, -0.8);
glEnd();

glutSwapBuffers();
```



توابع مفید و کاربردی در OpenGL

تعدادی از توابع سودمند و کاربردی مانند glEnable که به ناچار مجبور بودیم در برخی از مثال ها به کار ببریم را در اینجا توضیح خواهیم داد. این توابع حالت عمومی دارند و برای بسیاری از اشکال در OpenGL کاربرد دارند.

گرفتن مقادیر متغیرهای محیطی در OpenGL

با استفاده از توابع زیر می توانید مقدار فعلی متغیرهای محیطی را بدست آورید :

```
void glGetBooleanv(GLenum pname, GLboolean *params);

void glGetDoublev(GLenum pname, GLdouble *params);

void glGetFloatv(GLenum pname, GLfloat *params);
```

```
void glGetIntegerv(GLenum pname, GLint *params);
```

این توابع دقیقا کار مشابهی انجام می دهند و تنها تفاوت آن ها در نوع متغیر params است که تعدد این توابع به خاطر عدم پشتیبانی زبان C از سربارگذاری توابع است (OpenGL با C نوشته شده است) pname همان متغیر محیطی OpenGL است که می خواهید مقدار آن را بگیرید و params متغیری است که مقدار pname در آن قرار می گیرد. و دلیل اشاره گر بودن آن نیز به خاطر اینکه بعضی از متغیرهای محیطی آرایه ای و یا ساختاری هستند.

برای مثال در کد زیر مقدار فعلی اندازه ی نقاط در OpenGL را می گیریم :

```
GLfolat nowSize;

glGetFloatv(GL_POINT_SIZE, &nowSize);
```

نکته : توابع عمومی و مشابه با این توابع برای دادن مقادیر به متغیرهای محیطی وجود ندارد. بلکه برای هرکدام از آن ها توابع بخصوصی وجود دارد. برای مثال بالا تابع glPointSize داریم که بحث شد.

فعال و یا غیرفعال کردن قابلیت ها

قبلا با glEnable در مثال ها آشنا شده اید. هنگامی که بخواهید قابلیتی را اضافه کنید از glEnable و هنگامی که بخواهید آن را غیر فعال کنید از glDisable استفاده می کنیم. اما بهتر است قبل از اینکه قابلیتی را فعال و یا غیرفعال کنید ، وضعیت فعلی آن را تشخیص دهید. این کار با تابع glIsEnabled انجام پذیر است. در صورتی که قابلیت مورد نظر فعال باشد GL_TRUE و در غیر اینصورت GL_FALSE برگشت داده می شود. برای مثال در کد زیر ابتدا بررسی می شود که آیا ویژگی smooth (صاف کردن) خطوط فعال است یا خیر ، سپس آن را فعال می کنیم :

```
if( glIsEnabled(GL_LINE_SMOOTH) == GL_FALSE )
    glEnable(GL_LINE_SMOOTH);
```

گرفتن خطاهای OpenGL

با استفاده از تابع glGetError() می توانید انواع خطاهای زمان اجرای OpenGL را بدست آورید. مقدار بازگشی این تابع از نوع GLenum است که می توانید آن را در متغیری از این نوع ذخیره کنید. انواع خطاهای OpenGL به شرح زیر است :

- GL_NO_ERROR : هیچ خطایی وجود ندارد. بهتر است قبل از فرستادن برنامه به خروجی این خطا را بگیرید (!)
- GL_INVALID_ENUM : همان نوع متغیرها و ثوابت از پیش تعریف شده ی OpenGL است. در صورتی که نوع نادرستی انتخاب کرده باشید (معمولا برای ارسال به توابع) با این خطا مواجه خواهید شد.
- GL_INVALID_VALUE : هنگامی که مقدار فرستاده شده بیش از محدوده ی مجاز آن متغیر باشد رخ می دهد.
- GL_INVALID_OPERATION : هنگامی رخ می دهد که اعمالی را بر روی نوعی از داده ها انجام دهید که غیرقابل انجام باشند (مانند اعمال حسابی)
- GL_STACK_OVERFLOW : هنگامی رخ می دهد که تعداد ماتریس های فرستاده شده بیش از حد ظرفیت پشته باشد (در مورد ماتریس ها در OpenGL بعدا بحث خواهیم کرد)

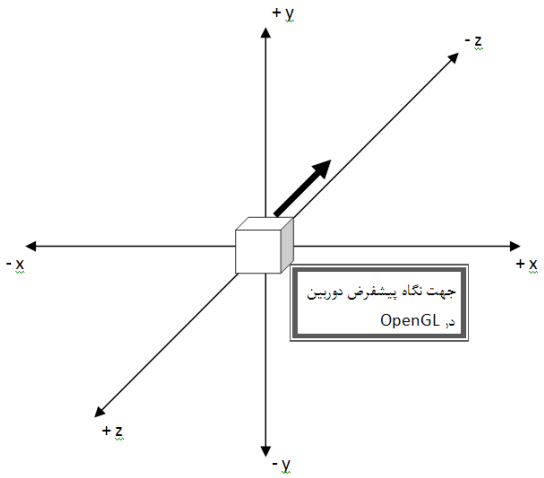
- GL_STACK_UNDERFLOW : هنگامی که برداشت ها از پشته بیش از تعداد موجود باشد این خطا رخ می دهد.
- GL_OUT_OF_MEMORY : هنگامی رخ می دهد که مقدار حافظه ی اختصاص داده شده به OpenGL توسط سیستم عامل کم باشد که این خطا بستگی به کد شما ندارد و بستگی به سیستم عامل میزبان دارد.

ورود به دنیای ۳–بعدی در OpenGL

مواردی که تاکنون بررسی کردیم حتی رئوسی هم که در مثال ها استفاده کردیم به صورت دوبعدی (glVertex2) دادیم. حال وارد دنیای سه بعدی می شویم. و اشکال پایه ای که بررسی کردیم را در سه بعد نیز رسم خواهیم کرد. همچنین به بررسی تعدادی از اشکال ۳-بعدی آماده در کتابخانه ی GLUT می پردازیم. اما قبل از همه ی این ها بایستی با نحوه ی مختصات ۳-بعدی در OpenGL آشنا شویم.

نحوه ی بیان مختصات ۳-بعدی در OpenGL

قبل از شروع برنامه نویسی در محیط ۳-بعدی بایستی نحوه ی نگاه OpenGL به مختصات ۳-بعدی را بدانید. شکل زیر این موضوع را به خوبی نشان می دهد.



برای رسم اشکال دو بعدی نیازی نداشتیم که بدانیم دوربین OpenGL (یا همان چشم ما) به چه نقطه ای از مختصات نگاه می کند. اما همان طور که در شکل می بینید به صورت پیشفرض دوربین به قسمت منفی محور z ها نگاه می کند. (یعنی اگر به طور مثال نقطه ای با مشخصات (۲,۴,۷) به سیستم بدهید ، از نگاه دوربین قابل رویت نخواهد بود ولی نقطه ی (۲,۴,-۷) به خوبی قابل مشاهده است) البته این جهت نگاه به محورها را در مراحل بعدی تغییر خواهیم داد.

بیان مختصات در ۳-بعد

برای بیان مختصات رئوس در سه بعد تنها کافی است مکان قرارگیری را در محور z به همراه محورهای دوبعدی x و y در نظر داشته و در تابع زیر جایگذاری کنید :

glVertex3{dfis}(مختصات سه بعدی راس)

همان طور که ملاحظه می کنید ، تفاوت چندانی با ۲-بعد وجود ندارد ، تنها زمانی که رئوس را مشخص می کنید بایستی به محل قرارگیری دوربین (قسمت قبل) توجه داشته باشید.

اشیای سه بعدی آماده در کتابخانه ی GLUT

نکته : قبل از رسم این اشکال بهتر است تابع glRotatef(40,1,0,1) را قبل از بلوک شروع و پایان قرار دهید تا بهتر بتوانید اشکال را ببینید. بعـدا بـه تفصیل در مورد این تابع بحث خواهیم کرد. (با کم یا زیاد کردن عدد ۴۰ می توانید چرخش کمتر یا بیشتری داشته باشید)

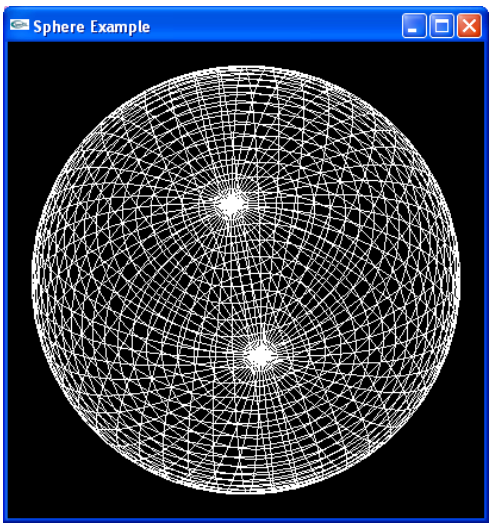
کره :

با استفاده از دو تابع زیر می توانید کره ی توپر و توخالی را در OpenGL رسم کنید :

```
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
```

که در آن ها radius شعاع کره ، slices تعداد تقسیمات حول محور z و stacks تعداد تقسیمات در طول محور z است.

مثالی از کره ی توخالی را در شکل زیر می بینید (radius=0.9, slices=50, stacks=40)



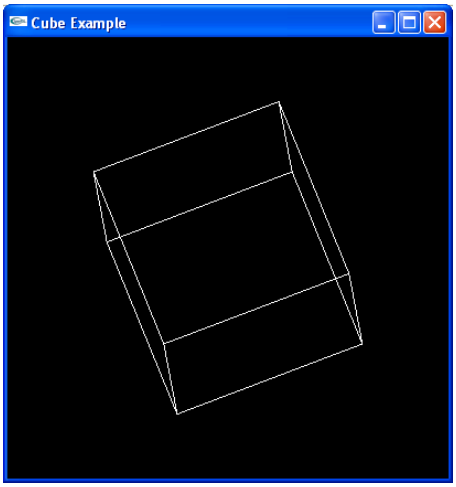
مکعب :

با استفاده از دو تابع زیر می توانید دو نوع مکعب توپر و توخالی رسم کنید :

```
void glutSolidCube(GLdouble size);
void glutWireCube(GLdouble size);
```

که size اندازه ی هرکدام از اضلاع را مشخص می کند.

در شکل زیر مثالی از کره ی توخالی با اندازه ی ضلع ۹/۰ را ملاحظه می کنید.



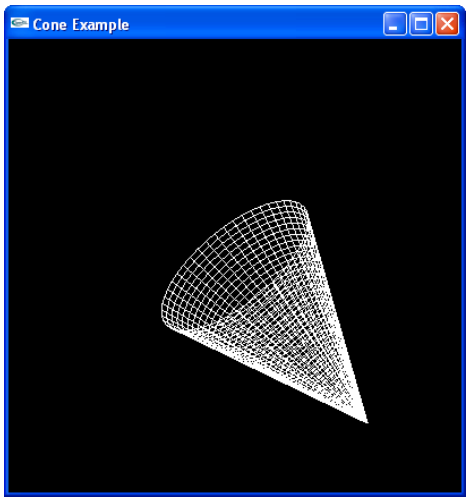
مخروط :

امکان ایجاد دو نوع مخروط توپر و توخالی با استفاده از توابع زیر :

```
void glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
void glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
```

base شعاع قاعده هرم ، height ارتفاع هرم ، slices تعداد تقسیمات حول محور z و stacks تعداد تقسیمات در طول محور z را مشخص می کند.

مثال زیر یک مخروط با مشخصات (base=0.4, height=1.0, slices=50, stacks=40) را نشان می دهد :



حلقه ی بسته (Torus)

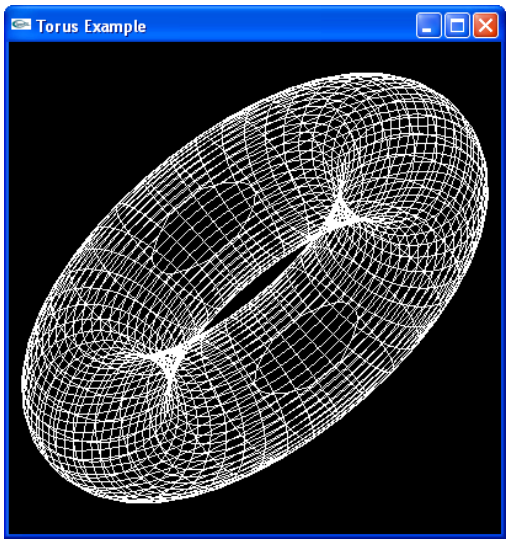
امکان ایجاد دو نوع از این شکل (توپر و توخالی) مانند بقیه ی اشکال وجود دارد. توابع آن به صورت زیر است :

```
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint
rings);
```

```
void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);
```

innerRadius شعاع داخلی ، outerRadius شعاع بیرونی ، nsides تعداد اضلاع برای هر بخش دایره ای و rings تعداد تقسیمات محـوری بـرای شکل است.

در شکل زیر مثالی با مشخصات (innerRadius=0.3, outerRadius=0.8, nsides=50, rings=40) می بینید :

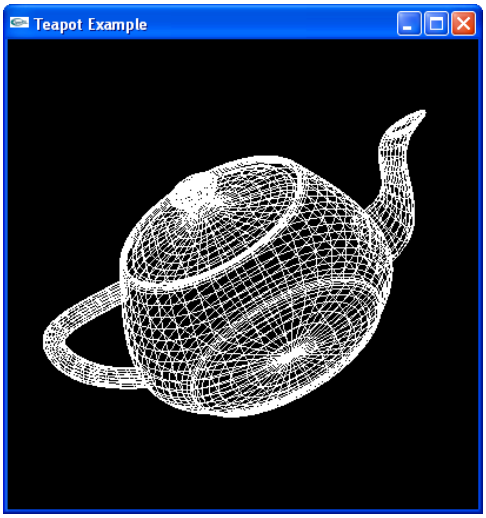


قوری :

این کتابخانه همچنین امکان رسم یک قوری (به صورت توپر و توخالی) را بیشتر جهت تست برنامه هایتان در اختیار می گذارد :

```
void glutSolidTeapot(GLdouble size);
void glutWireTeapot(GLdouble size);
```

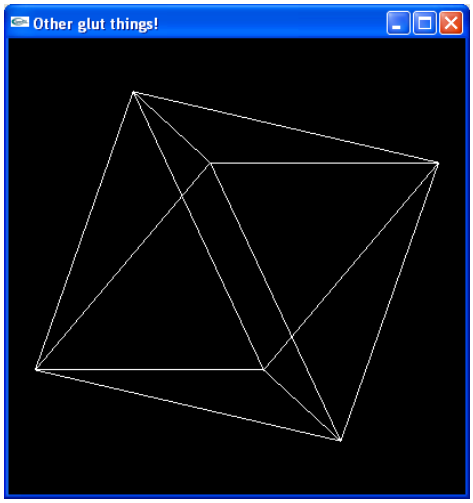
که پارامتر size اندازه ی قوری را مشخص می کند. در شکل زیر مثالی از قوری را با اندازه ی ۶/۰ ملاحظه می کنید :



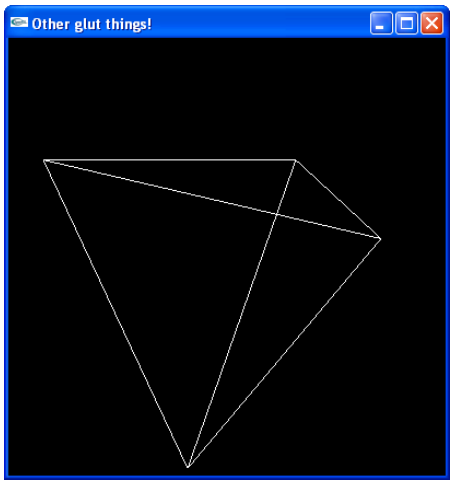
اشکال متفرقه ی دیگر (بدون آرگومان)

کتابخانه ی GLUT اشکال دیگری را در اختیارتان می گذارد که کنترلی بر آن ها (از لحاظ ارسال آرگومان به توابع آن ها) ندارید و فقط می توانید آن ها رسم کنید. این اشکال را به همراه تصویری از آن ها ملاحظه می کنید :

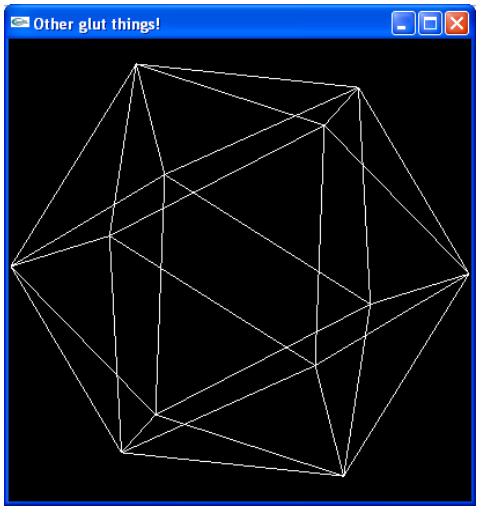
```
void glutSolidOctahedron(void);
void glutWireOctahedron(void);
```




```
void glutSolidTetrahedron(void);
void glutWireTetrahedron(void);
```




```
void glutSolidIcosahedron(void);
void glutWireIcosahedron(void);
```



رسم یک مکعب به صورت دستی :

اکنون که بیشتر با ۳-بعد آشنا شده اید و تعدادی شکل سه بعدی را رسم کردیم ، نوبت آن است که دانش قبلی در مورد اشکال ۲-بعدی را با سه بعد در آمیزیم و یک مکعب به صورت دستی (همان طور که دیدید مکعب به صورت آماده در کتابخانه ی glut موجود بود) را ایجاد کنیم.

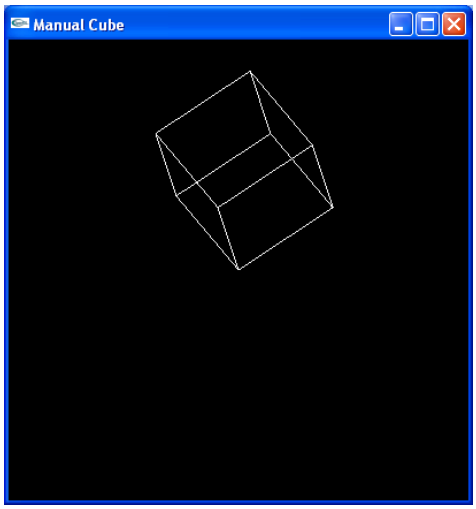
اشیای پایه ای که برای این کار نیاز داریم ، ۴-ضلعی (GL_QUADS) و چندتا خط (GL_LINES) است. مکعب را دقیقا همان طوری رسم می کنیم که بـر روی کاغذ انجام می دهیم. ابتدا دو مربع با فاصله از هم می کشیم و سپس با استفاده از خطوط محل تلاقی رئوس را به هم وصل می کنیم.

ابتدا کد زیر را ببینید (cpp.۰۴) :

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

2 int Height=400, Width=400;
3 float Size=0.5;

4 void display(void)
5 {
6     glClear(GL_COLOR_BUFFER_BIT);
7     glRotatef(50,1,0,1);

8     glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
9     glBegin(GL_QUADS);
10    glVertex3f(0.0, 0.0, 0.0);
11    glVertex3f(Size, 0.0, 0.0);
12    glVertex3f(Size, Size, 0.0);
13    glVertex3f(0.0, Size, 0.0);

14    glVertex3f(0.0, 0.0, -Size);
15    glVertex3f(Size, 0.0, -Size);
16    glVertex3f(Size, Size, -Size);
17    glVertex3f(0.0, Size, -Size);
18    glEnd();

19    glBegin(GL_LINES);
20    glVertex3f(0.0, Size, 0.0);
21    glVertex3f(0.0, Size, -Size);
```

```
22    glVertex3f(0.0, 0.0, 0.0);
23    glVertex3f(0.0, 0.0, -Size);

24    glVertex3f(Size, 0.0, 0.0);
25    glVertex3f(Size, 0.0, -Size);

26    glVertex3f(Size, Size, 0.0);
27    glVertex3f(Size, Size, -Size);
28    glEnd();

29    glutSwapBuffers();
30 }

31 int main(int argc, char **argv)
32 {
33    glutInit(&argc, argv);
34    glutInitDisplayMode(GLUT_DOUBLE);
35    glutInitWindowSize(Width, Height);
36    glutCreateWindow("Manual Cube");

37    glutDisplayFunc(display);

38    glutMainLoop();
39 }
```

همان فرم قبلی را برای برنامه ی خودمان حفظ کرده ایم. فقط در خط ۳ متغیری به نام Size تعریف کرده ایم تا در صورت نیاز اندازه ی مکعب را تغییر دهیم. در خط ۷ بازهم از تابع glRotate استفاده کرده ایم که طبق آرگومان های داده شده ، ۵۰ درجه در راستای محورهای x و z چرخش می کند (در بحث های بعدی به این موضوع می پردازیم). این کار را برای این کرده ایم که مکعب به طور کامل نمایش داده شود. در خط ۸ از تابع glPolygonMode استفاده کرده ایم تا از توپر شدن چهارضلعی ها جلوگیری کنیم (این تابع را قبلا توضیح داده ایم) در خطوط ۱۰ تا ۱۳ مربعی را در z=0 با طول ضلع Size رسم کرده ایم و در خطوط ۱۴ تا ۱۷ مربعی را در پشت آن با z=-Size (به علامت منفی توجه کنید چون گفتیم که با دادن z مثبت شکل محو خواهد شد) و طول ضلع Size رسم کرده ایم و سپس در خطوط ۱۹ تا ۲۸ محل تلاقی اضلاع را بوسیله ی خطوط به هم متصل کرده ایم. بقیه ی خطوط نیز همانند مثال های قبلی است. خروجی حاصل از این برنامه را در شکل زیر می بینید :



منبع اصلی : beginning OpenGL Game Programming نویسندگان :Dave Astle و Kevin Hawkings از THOMSON COURSE THECNOLOGY