



# OpenGL 编程

---

## 坐标变换

# 坐标变换

- 要观察一个物体，可以：
  - 从不同的位置去观察它（视图变换）
  - 移动或者旋转它，可以放大或缩小它（模型变换）
  - 如果把物体画下来，可以选择：是否需要一种“近大远小”的透视效果。另外，可能只希望看到物体的一部分（剪裁）（投影变换）
  - 希望把整个看到的图形画下来，但它只占据纸张的一部分，而不是全部（视口变换）

# 模型变换和视图变换

- 在OpenGL中通过矩阵乘法实现各种变换
- 改变观察点的位置与方向和改变物体本身的位置与方向具有等效性：
  - OpenGL中实现两种功能使用同样的函数
  - 在进行变换前，应先设置当前操作的矩阵为“模型视图矩阵”，设置的方法  
`glMatrixMode(GL_MODELVIEW)`
  - 在进行变换前把当前矩阵设置为单位矩阵  
`glLoadIdentity()`

# 模型变换和视图变换

- 模型和视图变换中的三个函数：
  - **glTranslate\***把当前矩阵和一个表示移动物体的矩阵相乘：三个参数分别表示了三个坐标上的位移值
  - **glRotate\***把当前矩阵和一个表示旋转物体的矩阵相乘：物体将绕着(0,0,0)到(x,y,z)的直线以逆时针旋转，**angle**表示旋转的角度
  - **glScale\***把当前矩阵和表示缩放物体的矩阵相乘：**x,y,z**分别表示在该方向上的缩放比例

# 模型变换和视图变换

- 改变观察点的位置，四个函数：
  - **gluLookAt**: 前三个参数表示观察点的位置；中间三个参数指定视线上的任意点；后三个参数代表从(0,0,0)到(x,y,z)的直线，表示了观察者认为的“上”方向
- 假设当前矩阵为单位矩阵  $((RT)_v) = (R(T_v))$ 
  - 实际上是先进行移动，然后进行旋转
- “先移动后旋转”和“先旋转后移动”得到的结果很可能不同。

# 投影变换

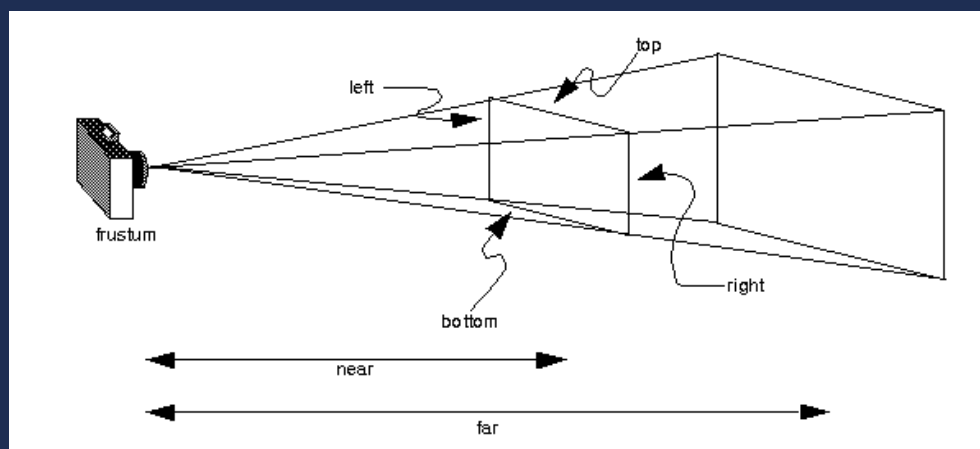
- 投影变换：定义一个可视空间，可视空间以外的物体不会被绘制到屏幕上
- OpenGL支持两种投影变换：透视投影和正投影
  - 变换前，应先设置当前操作的矩阵为投影变换视图矩阵，设置的方法  
`glMatrixMode(GL_PROJECTION);`
  - 变换前，把当前矩阵设置为单位矩阵  
`glLoadIdentity();`

# 投影变换

- 透视投影

- 产生结果如照片：近大远小
- `glFrustum`函数：将当前可视空间设置为透视投影空间

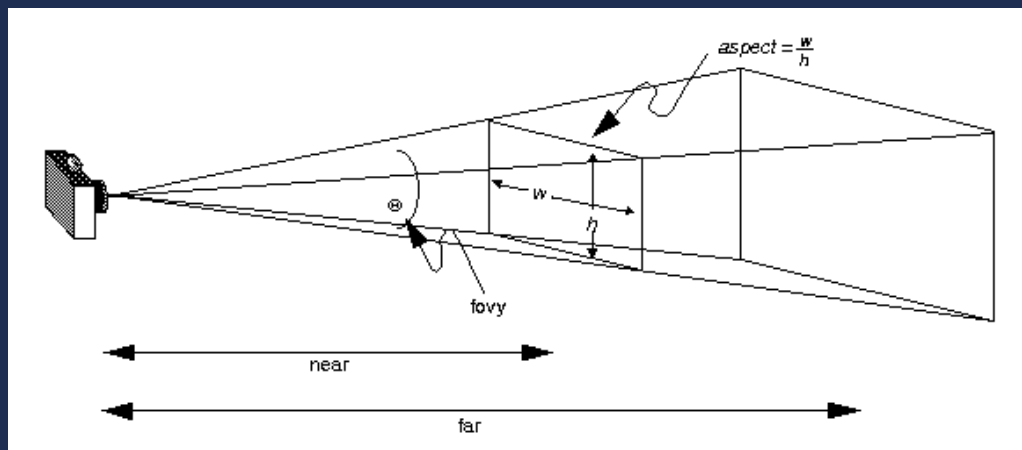
`void glFrustumf(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far);`



# 模型变换和视图变换

## – gluPerspective函数

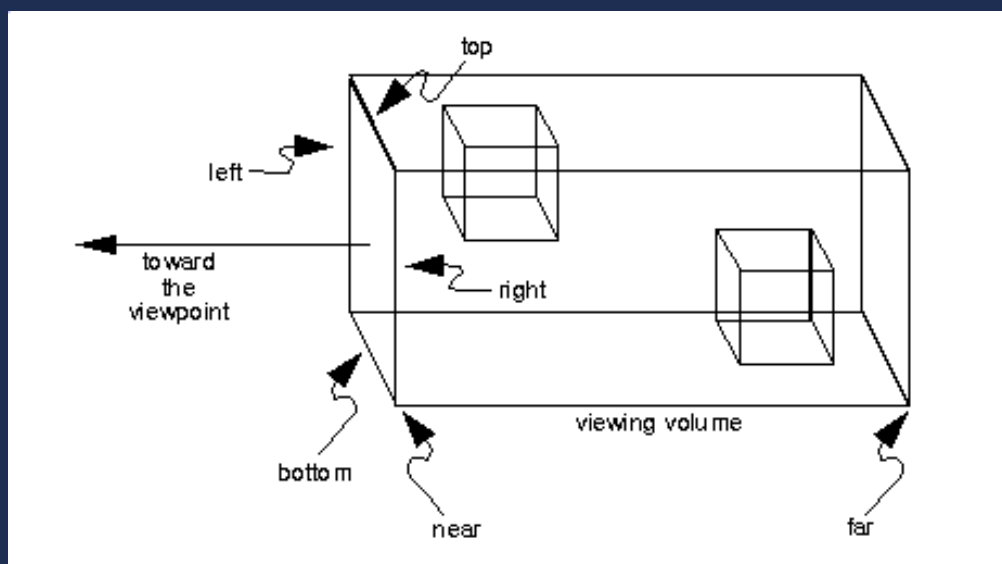
- fovy指定观察视角，一般使用45度
- aspect指定视区的高度和宽度比例，值为： $x(\text{width})/y(\text{height})$
- zNear,zfar指定近裁减面和远裁减面到观察者的距离(始终为正)





# 投影变换

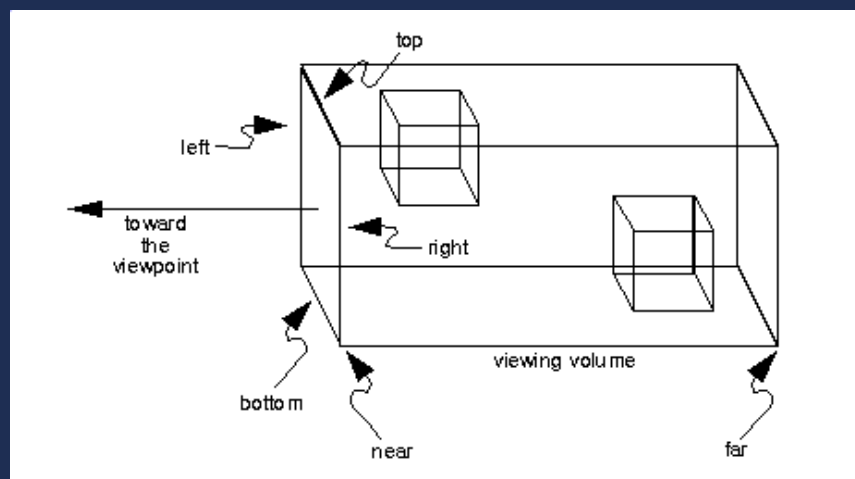
- 正投影：相当于无穷远处观察得到的结果，是一种理想状态
  - **glOrtho**函数：将当前的可视空间设置为正投影空间



# 模型变换和视图变换

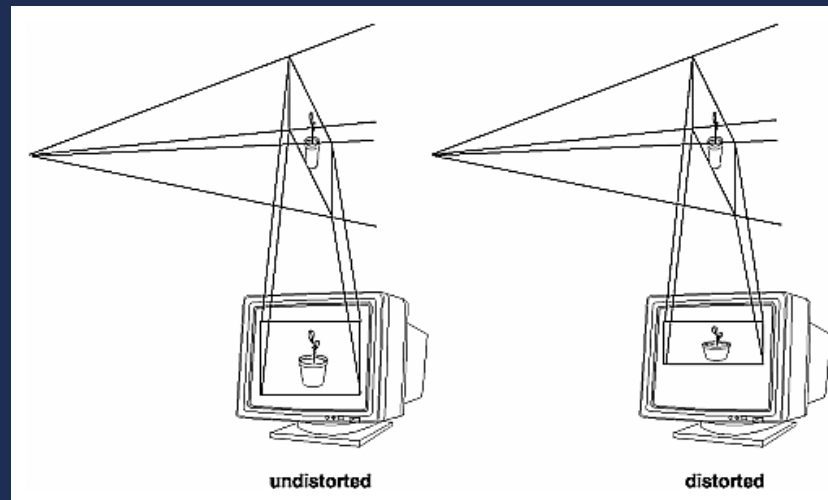
## — glOrtho函数

- **left, right**—指定左右裁减面的位置
- **bottom, top**—指定底、顶裁减面的位置
- **near, far**—指定近、远裁减面的位置
- 如图形本身是二维: glOrtho2D



# 视口变换

- 视口变换：应该把像素绘制到窗口什么位置？
  - **glViewport**: 两个参数定义了视口的左下角（0,0表示最左下方），后两个参数分别是宽度和高度。



# 操作矩阵堆栈

- 矩阵堆栈：替代先变换再逆变换
  - 进行矩阵操作时，有可能需要先保存某个矩阵，过一段时间再恢复它
    - 保存时，调用glPushMatrix函数
    - 取矩阵，调用glPopMatrix函数
    - 堆栈的容量至少可以容纳32个矩阵
    - 模型视图和投影矩阵都有相应的堆栈
- glMatrixMode来指定当前操作

# 举例

---

- 综合举例
  - 制作的是一个三维场景，包括了太阳、地球和月亮。



# 小结

- OpenGL通过矩阵变换来把三维物体转变为二维图象，并在屏幕上显示。
  - 指定当前操作是何种矩阵`glMatrixMode`
  - 移动、旋转观察点或者移动、旋转物体
  - 缩放物体`glScale*`
  - 定义可视空间：正投影or透视投影
  - 定义绘制到窗口的范围`glViewport`
  - 矩阵“堆栈”，方便进行保存和恢复



# OpenGL 编程

---

## 计算机动画

# 计算机动画

---

- 电影和动画的工作原理：
  - 快速的把看似连续的画面一幅幅的呈现在人们面前
  - 一旦每秒钟呈现的画面超过**24**幅，人们就会错以为它是连续的
  - 通常观看的电视，每秒播放**25**或**30**幅画面
  - 对于一个正常人来说，每秒**60~120**幅图画是比较合适的




# 计算机动画

- 假设某动画共有n幅画面，工作步骤：
  - 显示第1幅，等待一小段时间，直到下一个 $1/24$ 秒
  - .....
  - 显示第n幅画面，...
  - 结束
- 如果用C语言伪代码来描述这一过程：

```
for(i=0; i<n; ++i)
{
    DrawScene(i);
    Wait();
}
```

# 双缓冲技术

---

- 计算机上的动画与实际的动画有些不同：
  - 实际的动画都是先画好了，播放的时候直接拿出来显示就行
  - 计算机动画则是画一张，就拿出来一张，再画下一张，再拿出来
- 当所需要绘制的图形很简单，那么这样也没什么问题。
- 当图形比较复杂，绘制需要的时间较长，问题就会变得突出。

# 双缓冲技术

- 计算机想象成一个画图比较快的人
  - 假如直接在屏幕上画图且图形比较复杂，则可能在只画了某幅图的一半时就被观众看到
  - 后面虽把画补全了，但观众的眼睛却又没有反应过来，还停留在原来那个残缺的画面
- 观众有时看到完整的图象，有时却又只看到残缺的图象，这样就造成了屏幕的闪烁
- 如何解决这一问题呢？



# 双缓冲技术

---

- 设想有两块画板
  - 画图的人在旁边画，画好以后把他手里的画板与挂在屏幕上的画板相交换，观众就不会看到残缺的画了
- 双缓冲技术：在存储器（很有可能是显存）中开辟两块区域，一块作为发送到显示器的数据，一块作为绘画的区域，在适当的时候交换它们

# 双缓冲技术

---

- 双缓冲技术

- 由于交换两块内存区域实际上只需要交换两个指针，这一方法效率非常高，应用广泛
- 虽然绝大多数平台都支持双缓冲技术，但这一技术并不是OpenGL标准中的内容
- OpenGL为了保证更好的可移植性，允许在实现时不使用双缓冲技术
- 常用的PC都是支持双缓冲技术的



# 双缓冲技术

- 双缓冲技术
  - 启动双缓冲功能，最简单的办法就是使用GLUT工具包：`glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);`
    - GLUT\_SINGLE表示单缓冲
    - GLUT\_DOUBLE就是双缓冲
  - 每次绘制完成时，需要交换两个缓冲区
    - 如使用GLUT工具包，只要在绘制完成时简单的调用`glutSwapBuffers`函数

# 实现连续动画

- 把绘制动画的代码写成
  - ```
for(i=0; i<n; ++i)
{
    DrawScene(i);
    glutSwapBuffers();
    Wait();
}
```
- 这样做不太符合窗口系统的程序设计思路

# 实现连续动画

- 第一个OpenGL程序
  - main函数: `glutDisplayFunc(&myDisplay)`  
如需绘制窗口, 请调用`myDisplay`函数
  - 为什么不直接调用, 而采用这种做法: 程序无法确定究竟什么时候该绘制窗口
- Windows系统: 支持同时显示多个窗口
  - 假如你的程序窗口碰巧被别的窗口遮住, 后来用户又把原来遮住的窗口移开, 这时你的窗口需要重新绘制
  - 你无法知道这一事件发生的具体时间, 只好委托操作系统来办了



# 实现连续动画

- 既然DrawScene都可以交给操作系统来代办了，那让整个循环运行起来的工作是否也可以交给操作系统呢？
- 先前的思路
  - 绘制，等待一段时间。如去掉等待的时间，就变成了绘制，绘制，.....，不停的绘制。
- 改进后的思路
  - 资源是公用的，不能因为动画让其他工作都停下来。因此，需在CPU空闲的时间绘制。

# 实现连续动画

---

- 在CPU空闲的时间调用某一函数
  - **glutIdleFunc**函数
- GLUT还提供了一些别的函数，例如“在键盘按下时做某事”等
- 举例：太阳、地球和月亮，让地球和月亮自己动起来

# 垂直同步

---

- 代码运行时，出现一些异常现象：
  - CPU几乎都用上了，但运动速度很快，根本看不清楚
  - CPU使用率很低，根本就没有把空闲时间完全利用起来
- 如何防止出现这类现象
  - 牵涉到关于垂直同步的问题



# 垂直同步

- 显示器的刷新频率一般为60~120Hz
  - 如计算机绘制简单的画面，则一秒钟可以绘制成千上万次
  - 如最大限度的利用计算机的处理能力，绘制很多幅画面，由于显示器的刷新频率，不仅造成性能的浪费，还可能带来一些负面影响
    - 例如：显示器只刷新到一半时，需绘制的内容却变化了，由于显示器是逐行刷新，于是显示器上下部分实际上是来自两幅画面

# 垂直同步

- 垂直同步技术：只有在显示器刷新时，才把绘制好的图象传输出去供显示
  - 计算机就不必去绘制大量的根本就用不到的图象了
  - 如显示器的刷新率为**NHz**，则计算机一秒钟只需要绘制**N**幅图象就足够，如果场景足够简单，就会造成比较多的**CPU**空闲
  - 几乎所有的显卡都支持“垂直同步”功能

# 垂直同步

- 引入的问题
  - 如刷新频率为60Hz,
    - 如绘制一幅图画比较短，则为60FPS
    - 如绘制一幅图画的时间 $1/50s$ ，则为30FPS
    - 如果每一幅图画的复杂程度是不一致的，将造成了帧速的跳动
- 使用场合：如果使用了大量的CPU而且速度很快无法看清
- 其它方法也可控制帧速

# 垂直同步

---

- 设置垂直同步开关的代码如下：
  - `wglSwapIntervalEXT(1);` //打开垂直同步，限制帧率
  - `wglSwapIntervalEXT(0);` //关闭垂直同步，充分发挥显卡的渲染能力

# 计算帧速

---

- 3D Mark软件
  - 可运行各种场景，测出帧速，并为系统评分
- 帧速
  - 一秒钟内播放的画面数目（**FPS**）
- 理论上，可测量绘制两幅画面之间时间 $t$ ，取倒数即为帧速
  - C语言中的**time**，**clock**等函数精度均有限



# 计算帧速

---

- 纸张测厚度思想
  - 用很多张纸叠在一起测厚度，计算平均值
- 平均帧速测试
  - 测量绘制**N**幅画面（包括垂直同步等因素的等待时间）需要的时间
  - 计算得到**FPS**的平均值



# 小结

---

- OpenGL动画
- 双缓冲技术
- 利用CPU空闲的时候绘制动画glutIdleFunc
- 垂直同步
- 简单的计算帧速（FPS）的方法



