逻辑教育
Logic education

Hello CC

OpenGL 主题【3】

视觉班—OpenGL 渲染技巧

课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

课堂目标:

1. 渲染过程产生的问题
2. 油画渲染
3. 正面&背面剔除
4. 深度测试
5. 多边形模型
6. 多边形偏移
7. 裁剪
8. 颜色混合

课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

课后作业:

1. 请在个人博客上更新一篇博文,选题如下:
    01. 谈谈图形图像渲染中的深度缓冲区
    02. 阐述隐藏面消除解决方案
    03. 阐述深度缓冲区带来的隐患以及预防方案

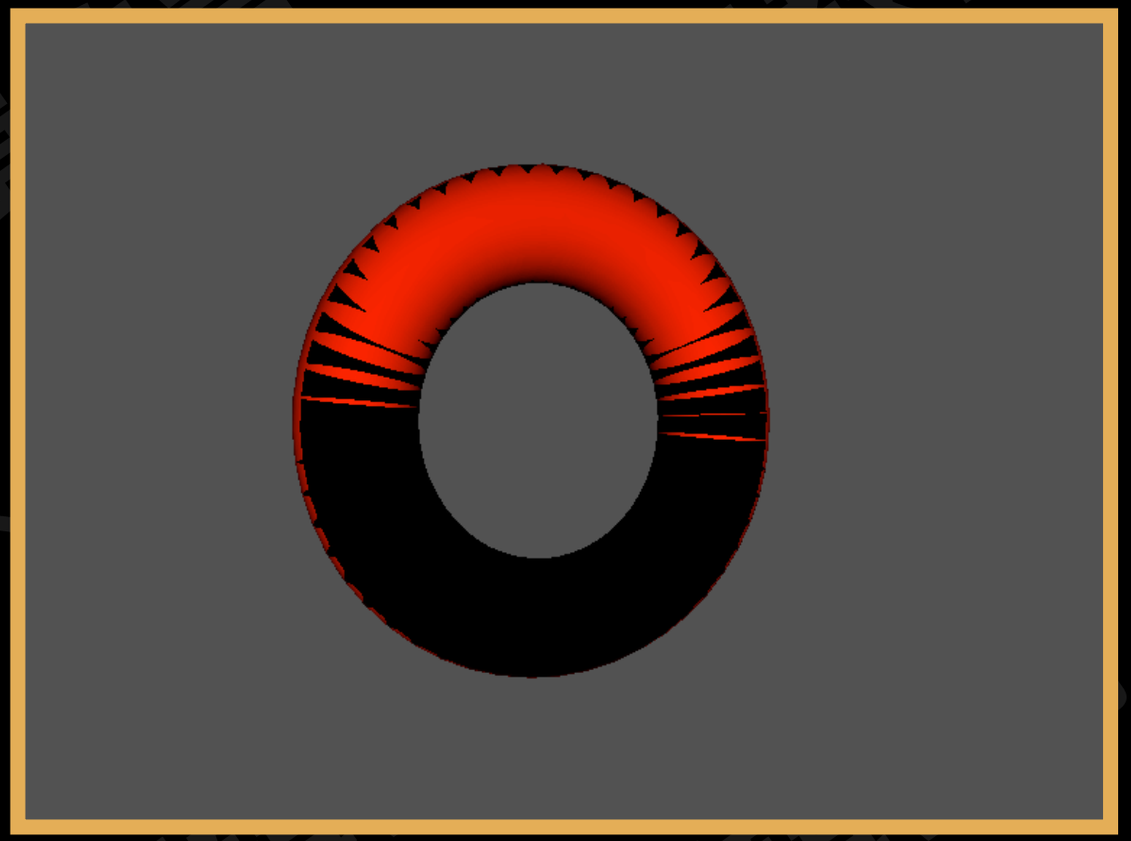
要求:

01.将课程内容加上自己的理解
02.更新的博客地址发送到讨论群.互相学习

课程研发:CC老师
课程授课:CC老师

在渲染过程中可能产生的问题

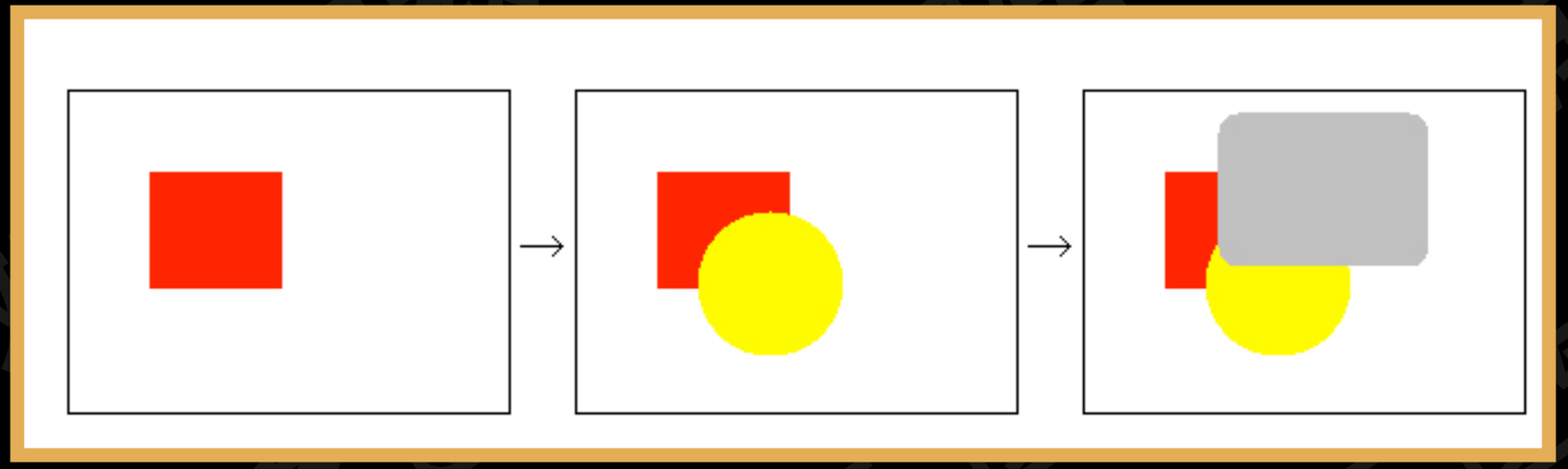
在绘制3D场景的时候,我们需要决定哪些部分是对观察者可见的,或者哪些部分是对观察者不可见的.对于不可见的部分,应该及早丢弃.例如在一个不透明的墙壁后,就不应该渲染.这种情况叫做”隐藏面消除”(Hidden surface elimination).



课程研发:CC老师
课程授课:CC老师

解决方案: 油画算法

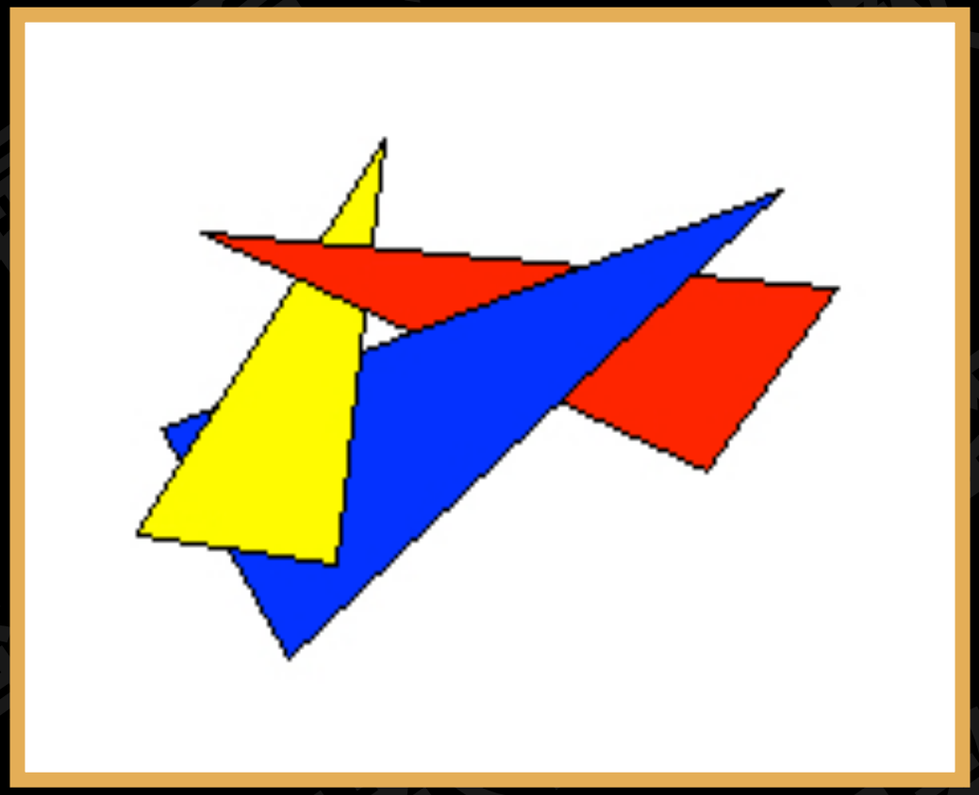
- 油画算法
  - ★ 先绘制场景中的离观察者较远的物体,再绘制较近的物体.
  - ★ 例如下面的图例: 先绘制红色部分,再绘制黄色部分,最后再绘制灰色部分,即可解决隐藏面消除的问题



课程研发:CC老师
课程授课:CC老师

解决方案：油画法弊端

- 油画算法
  - ★ 使用油画算法,只要将场景按照物理距离观察者的距离远近排序,由远及近的绘制即可.那么会出现什么问题? 如果三个三角形是叠加的情况,油画算法将无法处理.



课程研发:CC老师
课程授课:CC老师

解决方案: 正背面剔除(Face Culling)

- 背景
  - ★ 尝试相信一个3D图形,你从任何一个方向去观察,最多可以看到几个面?
  - ★ 答案是,最多3面. 从一个立方体的任意位置和方向上看,你用过不可能看到多于3个面.
  - ★ 那么思考? 我们为何要多余的去绘制那根本看不到的3个面?
  - ★ 如果我们能以某种方式去丢弃这部分数据,OpenGL 在渲染的性能即可提高超过50%.

课程研发:CC老师
课程授课:CC老师

解决方案: 正背面剔除(Face Culling)

- 解决问题
  - 如何知道某个面在观察者的视野中不会出现?
  - 任何平面都有2个面,正面/背面.意味着你一个时刻只能看到一面.

  - OpenGL 可以做到检查所有正面朝向观察者的面,并渲染它们.从而丢弃背面朝向的面. 这样可以节约片元着色器的性能.

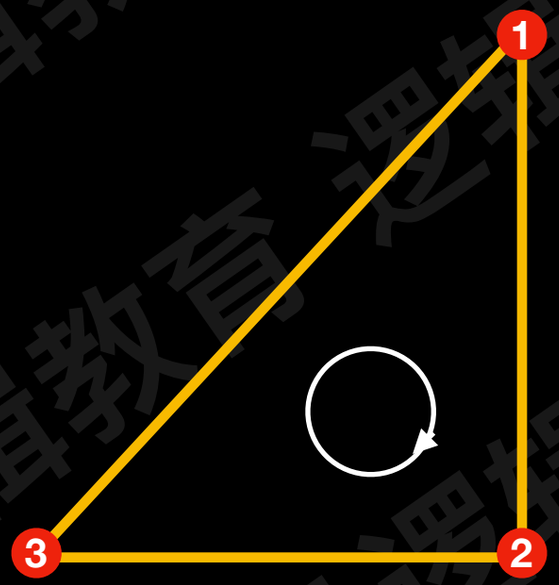
课程研发:CC老师
课程授课:CC老师

解决方案: 正背面剔除(Face Culling)

- 解决问题
  - 如果告诉OpenGL 你绘制的图形,哪个面是正面,哪个面是背面?
  - 答案: 通过分析顶点数据的顺序

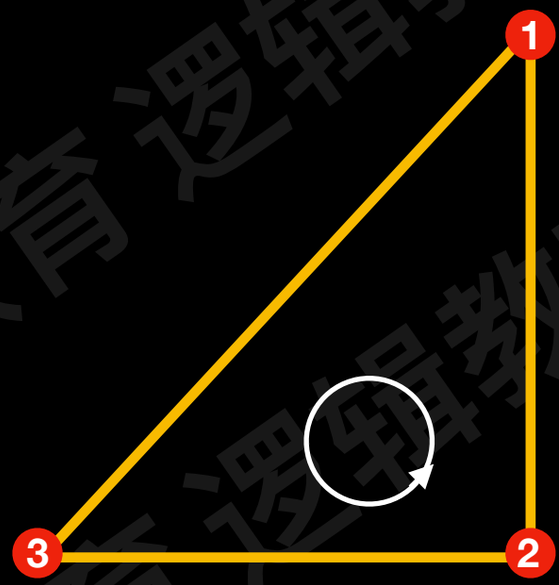
课程研发:CC老师
课程授课:CC老师

解决方案：分析顶点顺序



顺时针(Clockwise)
1—> 2—> 3

```
GLfloat vertices[] = {
    //顺时针
    vertices[0], // vertex 1
    vertices[1], // vertex 2
    vertices[2], // vertex 3
    // 逆时针
    vertices[0], // vertex 1
    vertices[2], // vertex 3
    vertices[1] // vertex 2
};
```



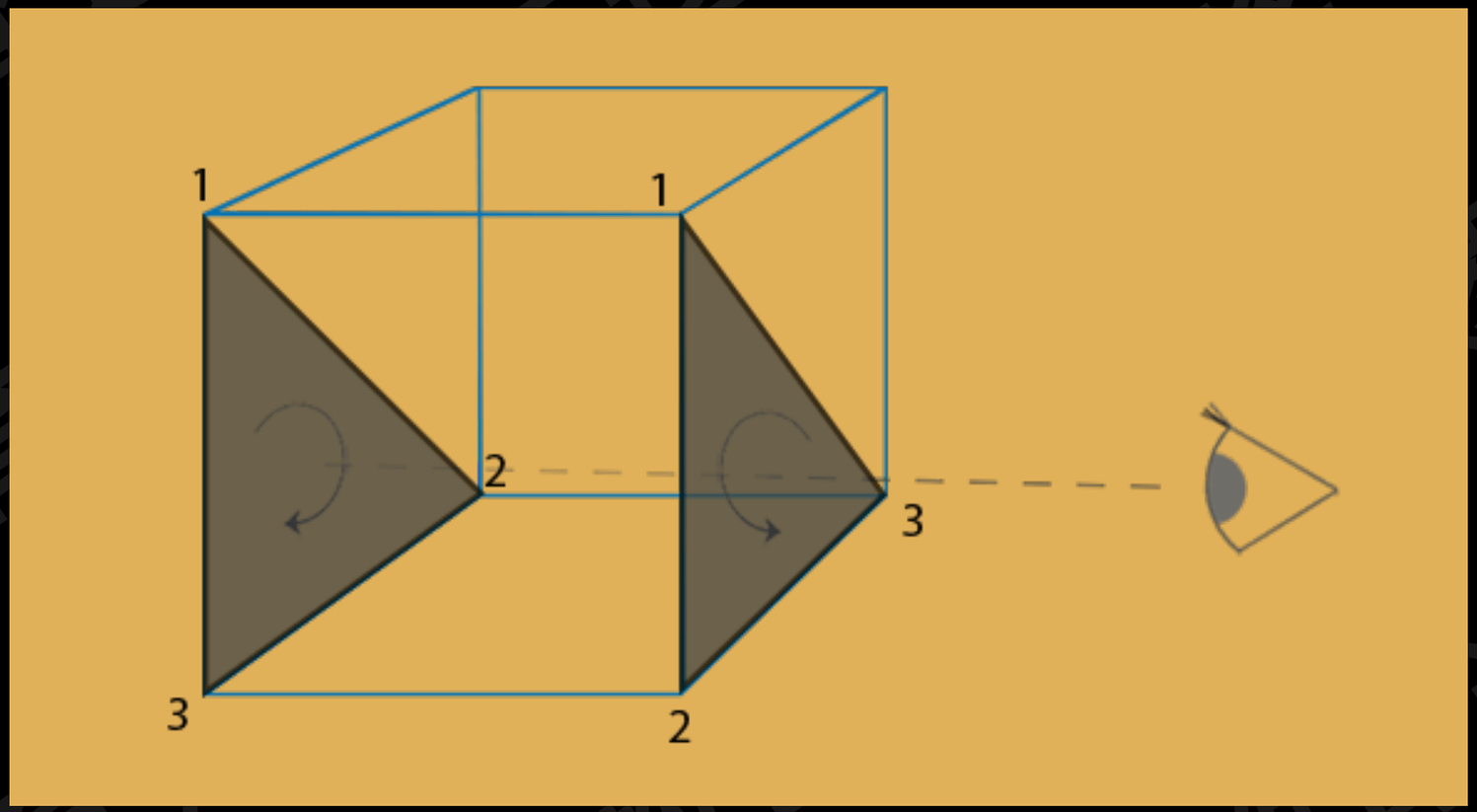
逆时针(Counter-clockwise)
1—> 3—> 2

- 正面/背面区分
  - 正面: 按照逆时针顶点连接顺序的三角形面
  - 背面: 按照顺时针顶点连接顺序的三角形面

课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

解决方案: 分析立方体中的正背面



- 分析
  - 左侧三角形顶点顺序为: 1—> 2—> 3 ; 右侧三角形的顶点顺序为: 1—> 2—> 3 .
  - 当观察者在右侧时,则右边的三角形方向为逆时针方向则为正面,而左侧的三角形为顺时针则为.面
  - 当观察者在左侧时,则左边的三角形为逆时针方向判定为正面,而右侧的三角形为顺时针判定为背面.
- 总结
  - 正面和背面是有三角形的顶点定义顺序和观察者方向共同决定的.随着观察者的角度方向的改变,正面背面也会跟着改变

课程研发:CC老师
课程授课:CC老师

1.6 解决方案：分析立方体中的正背面

- 开启表面剔除(默认背面剔除)

  void glEnable(GL_CULL_FACE);

- 关闭表面剔除(默认背面剔除)

  void glDisable(GL_CULL_FACE);

- 用户选择剔除那个面(正面/背面)

  void glCullFace(GLenum mode);
  mode参数为: GL_FRONT,GL_BACK,GL_FRONT_AND_BACK ,默认GL_BACK

- 用户指定绕序那个为正面

  void glFrontFace(GLenum mode);
  mode参数为: GL_CW,GL_CCW,默认值:GL_CCW

- 例如,剔除正面实现(1)

  glCullFace(GL_BACK);
  glFrontFace(GL_CW);

- 例如,剔除正面实现(2)

  glCullFace(GL_FRONT);

课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

了解深度

- 什么是深度?
  - ★ 深度其实就是该像素点在3D世界中距离摄像机的距离,Z值
- 什么是深度缓冲区?
  - ★ 深度缓存区,就是一块内存区域,专门存储着每个像素点(绘制在屏幕上的)深度值.深度值(Z值)越大,则离摄像机就越远.
- 为什么需要深度缓冲区?
  - ★ 在不使用深度测试的时候,如果我们先绘制一个距离比较近的物理,再绘制距离较远的物理,则距离远的位图因为后绘制,会把距离近的物体覆盖掉. 有了深度缓冲区后,绘制物体的顺序就不那么重要的. 实际上,只要存在深度缓冲区,OpenGL 都会把像素的深度值写入到缓冲区中. 除非调用glDepthMask(GL_FALSE).来禁止写入.

课程研发:CC老师
课程授课:CC老师

解决方法: Z-buffer方法(深度缓冲区Depth-buffer)

- 深度测试
  - 深度缓冲区(DepthBuffer)和颜色缓存区(ColorBuffer)是对应的.颜色缓存区存储像素的颜色信息,而深度缓冲区存储像素的深度信息. 在决定是否绘制一个物体表面时, 首先要将表面对应的像素的深度值与当前深度缓冲区中的值进行比较. 如果大于深度缓冲区中的值,则丢弃这部分.否则利用这个像素对应的深度值和颜色值.分别更新深度缓冲区和颜色缓存区. 这个过程称为”深度测试”

深度值计算

- 深度值计算
  - 深度值一般由16位，24位或者32位值表示，通常是24位。位数越高的话，深度的精确度越好。深度值的范围在[0,1]之间，值越小表示越靠近观察者，值越大表示远离观察者。
  - 深度缓冲主要是通过计算深度值来比较大小，在深度缓冲区中包含深度值介于0.0和1.0之间，从观察者看到其内容与场景中的所有对象的 z 值进行了比较。这些视图空间中的 z 值可以在投影平头截体的近平面和远平面之间的任何值。我们因此需要一些方法来转换这些视图空间 z 值到 [0，1] 的范围内,下面的 (线性) 方程把 z 值转换为 0.0 和 1.0 之间的值：

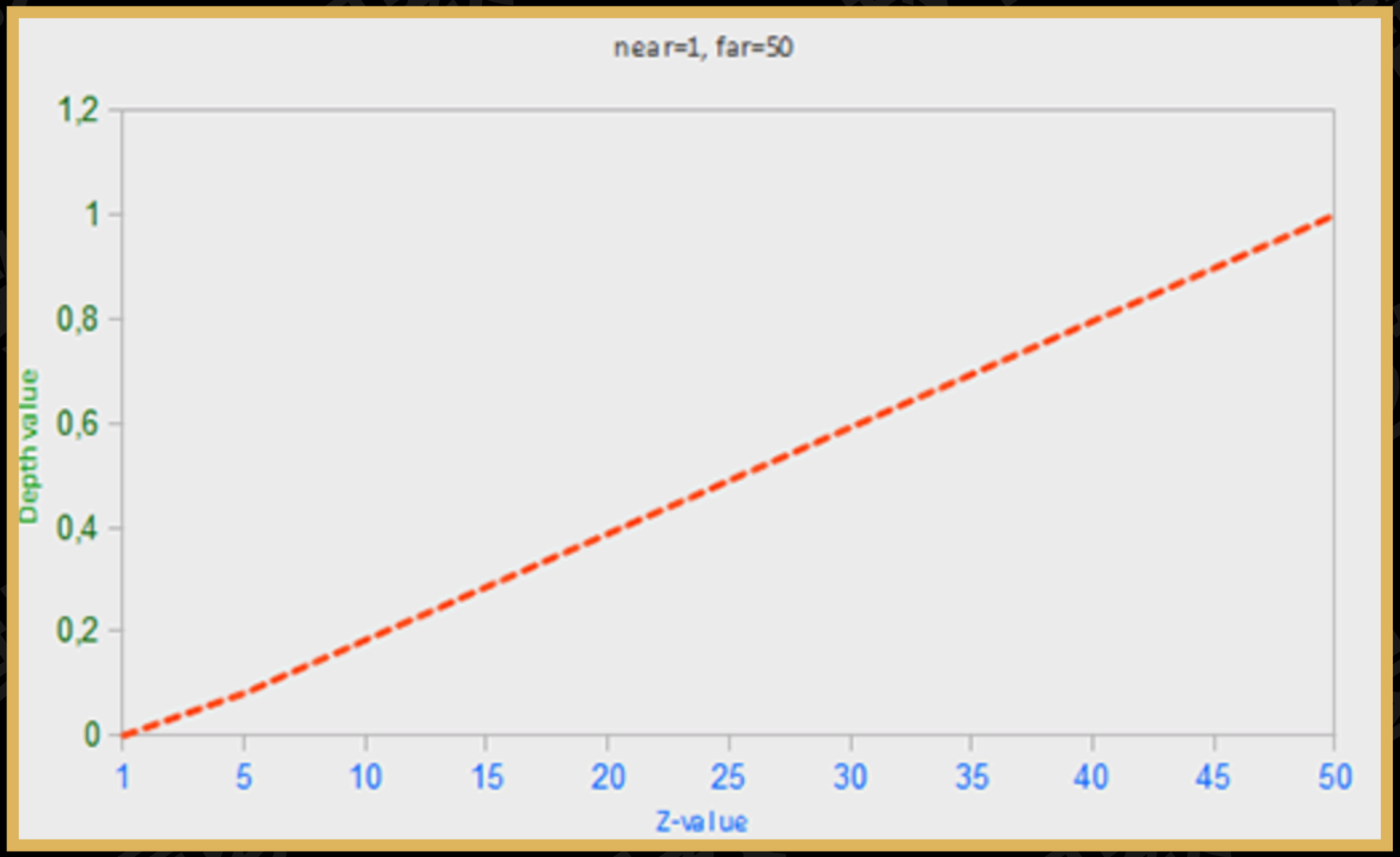
$$F_{depth} = \frac{z - near}{far - near}$$

far和near是提供到投影矩阵设置可见视图截锥的远近值

课程研发:CC老师
课程授课:CC老师

拓展知识点: 非线性深度缓存

- 方程带内锥截体的深度值 z，并将其转换到 [0，1] 范围。在下面的图给出 z 值和其相应的深度值的关系:



课程研发:CC老师
课程授课:CC老师

拓展知识点: 非线性深度缓存

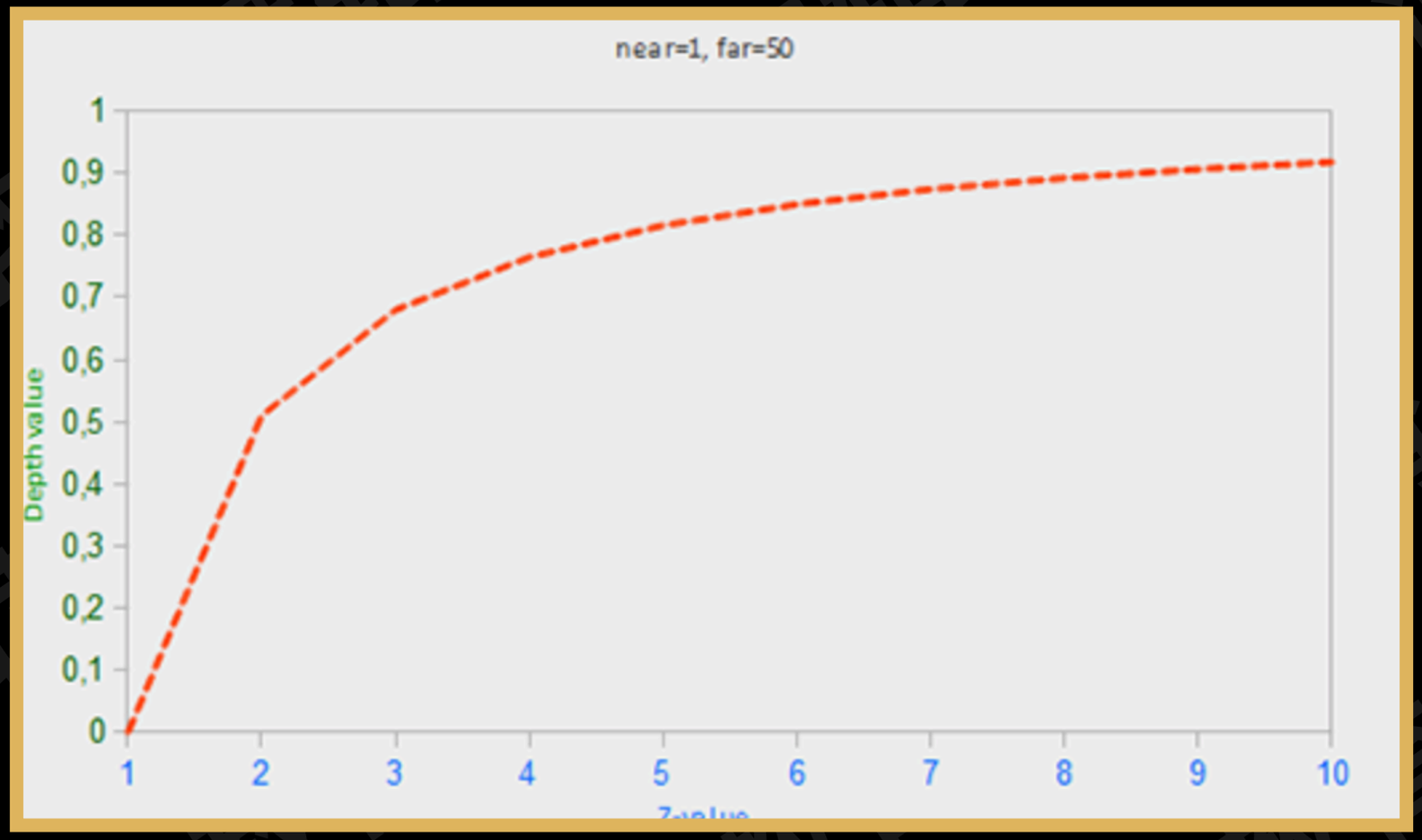
- 在实践中是可以减少使用这样的线性深度缓冲区。正确的投影特性的非线性深度方程是和1/z成正比的 ，由于非线性函数是和 1/z 成正比，例如1.0 和 2.0 之间的 z 值，将变为 1.0 到 0.5之间， 这样在z非常小的时候给了我们很高的精度。方程如下所示

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

课程研发:CC老师
课程授课:CC老师

拓展知识点: 非线性深度缓存

- 要记住的重要一点是在深度缓冲区的值不是线性的屏幕空间 (它们在视图空间投影矩阵应用之前是线性)。值为 0.5 在深度缓冲区并不意味着该对象的 z 值是投影平头截体的中间;顶点的 z 值是实际上相当接近近平面!你可以看到 z 值和产生深度缓冲区的值在下列图中的非线性关系



课程研发:CC老师
课程授课:CC老师

拓展知识点: 非线性深度缓存

屏幕空间的深度值是非线性如他们在z很小的时候有很高的精度，较大的 z 值有较低的精度。该片段的深度值会迅速增加，所以几乎所有顶点的深度值接近 1.0。如果我们小心的靠近物体，你最终可能会看到的色彩越来越暗，意味着它们的 z 值越来越小，这清楚地表明深度值的非线性特性。近的物体相对远的物体对的深度值比对象较大的影响。只移动几英寸就能让暗色完全变亮。

但是我们可以让深度值变换回线性。要实现这一目标我们需要让点应用投影变换逆的逆变换，成为单独的深度值的过程。这意味着我们必须首先重新变换范围 [0，1] 中的深度值为单位化的设备坐标(normalized device coordinates)范围内 [-1，1] (裁剪空间(clip space))。然后，我们想要反转非线性方程：

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

课程研发:CC老师
课程授课:CC老师

拓展知识点: 非线性还原线性[0,1]

```
//片元着色器
out vec4 color;

float LinearizeDepth(float depth)
{
    float near = 0.1;
    float far = 100.0;
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near) / (far + near - z * (far - near));
}

void main()
{
    float depth = LinearizeDepth(gl_FragCoord.z);
    color = vec4(vec3(depth), 1.0f);

}
```

课程研发:CC老师
课程授课:CC老师

使用深度测试

- 深度缓冲区,一般由窗口管理系统,GLFW创建.深度值一般由16位,24位,32位值表示. 通常是24位.位数越高,深度精确度更好.
- 开启深度测试
  glEnable(GL_DEPTH_TEST);
- 在绘制场景前,清除颜色缓存区,深度缓冲
  glClearColor(0.0f,0.0f,0.0f,1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
- 清除深度缓冲区默认值为1.0,表示最大的深度值,深度值的范围为(0,1)之间. 值越小表示越靠近观察者,值越大表示越远离观察者

课程研发:CC老师
课程授课:CC老师

指定深度测试判断式

```
//指定深度测试判断模式
void glDepthFunc(GLEnum mode);
```

函数	说明
GL_ALWAYS	总是通过测试
GL_NEVER	总是不通过测试
GL_LESS	在当前深度值 < 存储的深度值时通过
GL_EQUAL	在当前深度值 = 存储的深度值时通过
GL_LEQUAL	在当前深度值 <= 存储的深度值时通过
GL_GREATER	在当前深度值 > 存储的深度值时通过
GL_NOTEQUAL	在当前深度值 不等于 存储的深度值时通过
GL_GEQUAL	在当前深度值 >= 存储的深度值时通过

逻辑教育
Logic education

打开/阻断 深度缓存区写入

void glDepthMask(GLBool value);
value : GL_TURE 开启深度缓冲区写入; GL_FALSE 关闭深度缓冲区写入

课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

课堂案例

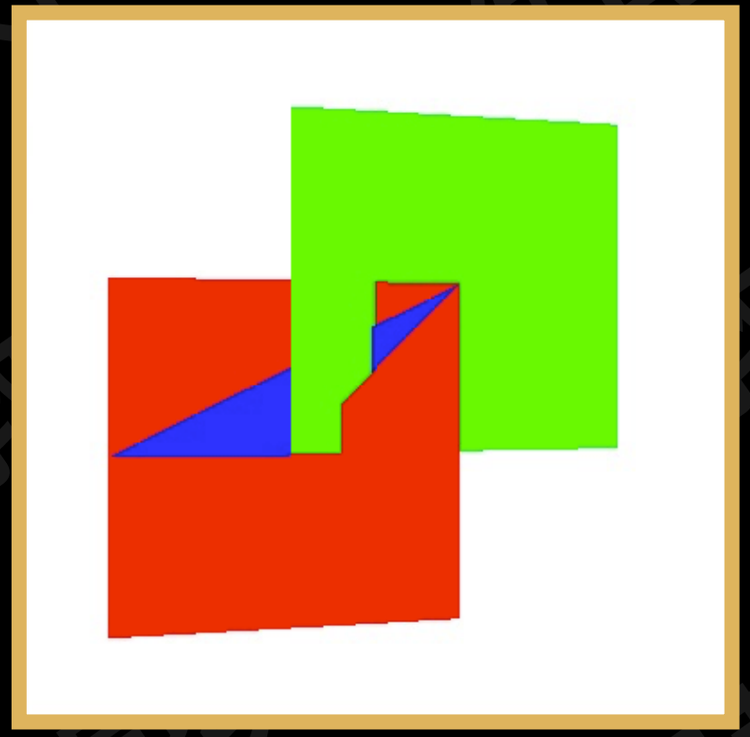
- 使用正面/背面剔除法和深度测试法来解决渲染效率问题.

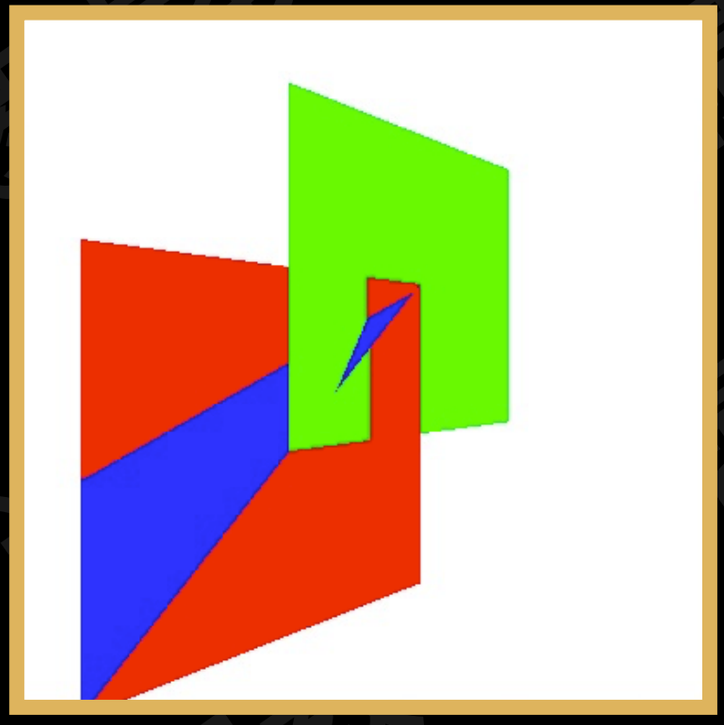
课程研发:CC老师
课程授课:CC老师

ZFighting闪烁问题的原因

- 为什么会出现 ZFighting 闪烁问题

  - 因为开启深度测试后,OpenGL 就不会再去绘制模型被遮挡的部分. 这样实现的显示更加真实.但是由于深度缓冲区精度的限制对于深度相差非常小的情况下.(例如在同一平面上进行2次制),OpenGL 就可能出现不能正确判断两者的深度值,会导致深度测试的结果不可预测.显示出来的现象时交错闪烁.的前面2个画面,交错出现.





课程研发:CC老师
课程授课:CC老师

ZFighting闪烁问题问题解决

- 第一步: 启用 Polygon Offset 方式解决

  - ★ 解决方法: 让深度值之间产生间隔.如果2个图形之间有间隔,是不是意味着就不会产生干涉.可以理解为在执行深度测试前将立方体的深度值做一些细微的增加.于是就能将重叠的2个图形深度值之前有所区分.

```
//启用Polygon Offset 方式
glEnable(GL_POLYGON_OFFSET_FILL)

参数列表:
    GL_POLYGON_OFFSET_POINT   对应光栅化模式: GL_POINT
    GL_POLYGON_OFFSET_LINE    对应光栅化模式: GL_LINE
    GL_POLYGON_OFFSET_FILL    对应光栅化模式: GL_FILL
```

课程研发:CC老师
课程授课:CC老师

ZFighting闪烁问题问题解决

- 第二步: 指定偏移量

★ 通过glPolygonOffset 来指定.glPolygonOffset 需要2个参数: factor , units
★ 每个Fragment 的深度值都会增加如下所示的偏移量:
Offset = (m * factor) + (r * units);
m : 多边形的深度的斜率的最大值,理解一个多边形越是与近裁剪面平行,m 就越接近于0.
r : 能产生于窗口坐标系的深度值中可分辨的差异最小值.r 是由具体是由具体OpenGL 平台指定的一个常量.
★ 一个大于0的Offset 会把模型推到离你(摄像机)更远的位置,相应的一个小于0的Offset 会把模型拉近
★ 一般而言,只需要将-1.0 和 -1 这样简单赋值给glPolygonOffset 基本可以满足需求.

```
void glPolygonOffset(GLfloat factor,GLfloat units);

应用到片段上总偏移计算方程式:
 Depth Offset = (DZ * factor) + (r * units);

DZ:深度值（Z值）
r:使得深度缓冲区产生变化的最小值

负值，将使得z值距离我们更近，而正值，将使得z值距离我们更远，
对于上节课的案例，我们设置factor和units设置为-1，-1
```

课程研发:CC老师
课程授课:CC老师

ZFighting闪烁问题问题解决

- 第三步: 关闭Polygon Offset

```
glDisable(GL_POLYGON_OFFSET_FILL)
```

课程研发:CC老师
课程授课:CC老师

ZFighting闪烁问题预防

- 不要将两个物体靠的太近，避免渲染时三角形叠在一起。这种方式要求对场景中物体插入一个少量的偏移，那么就可能避免ZFighting现象。例如上面的立方体和平面问题中，将平面下移0.001f就可以解决这个问题。当然手动去插入这个小的偏移是要付出代价的。
- 尽可能将近裁剪面设置得离观察者远一些。上面我们看到，在近裁剪平面附近，深度的精确度是很高的，因此尽可能让近裁剪面远一些的话，会使整个裁剪范围内的精确度变高一些。但是这种方式会使离观察者较近的物体被裁减掉，因此需要调试好裁剪面参数。
- 使用更高位数的深度缓冲区，通常使用的深度缓冲区是24位的，现在有一些硬件使用使用32位的缓冲区，使精确度得到提高

课程研发:CC老师
课程授课:CC老师

裁剪

在OpenGL 中提高渲染的一种方式.只刷新屏幕上发生变化的部分.OpenGL 允许将要进行渲染的窗口只去指定一个裁剪框.

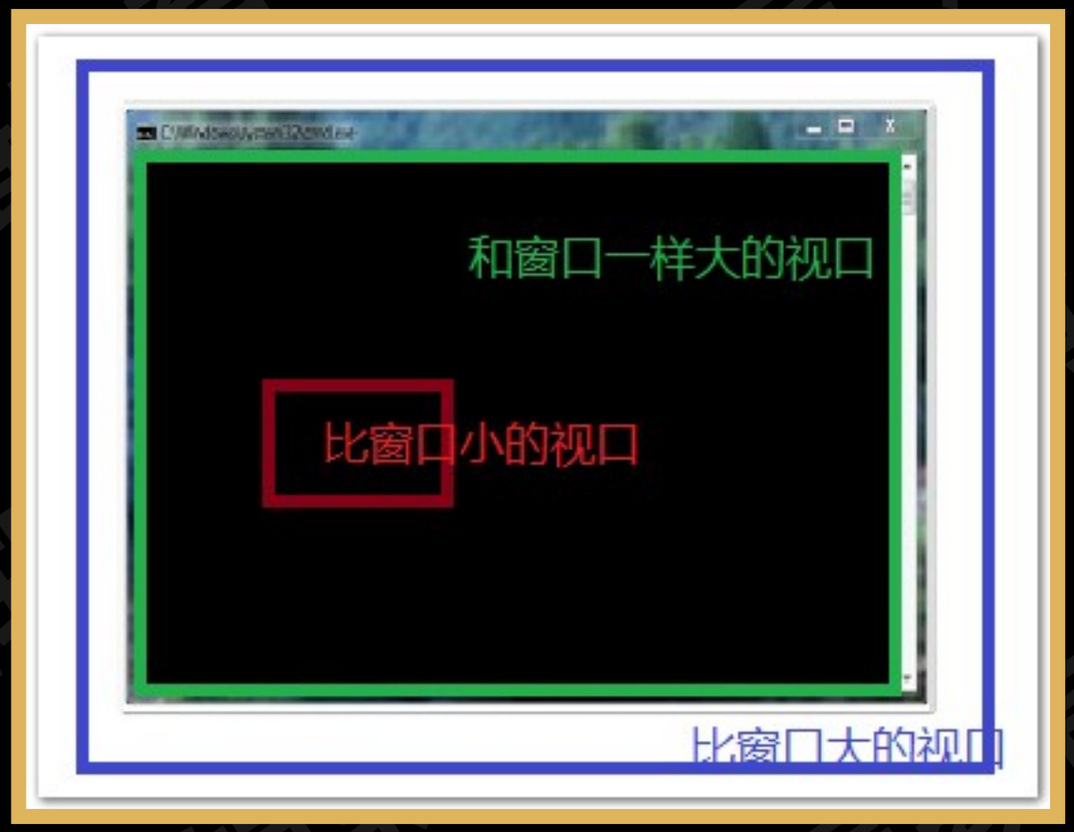
基本原理：用于渲染时限制绘制区域，通过此技术可以再屏幕（帧缓冲）指定一个矩形区域。启用剪裁测试之后，不在此矩形区域内的片元被丢弃，只有在此矩形区域内的片元才有可能进入帧缓冲。因此实际达到的效果就是在屏幕上开辟了一个小窗口，可以再其中进行指定内容的绘制。

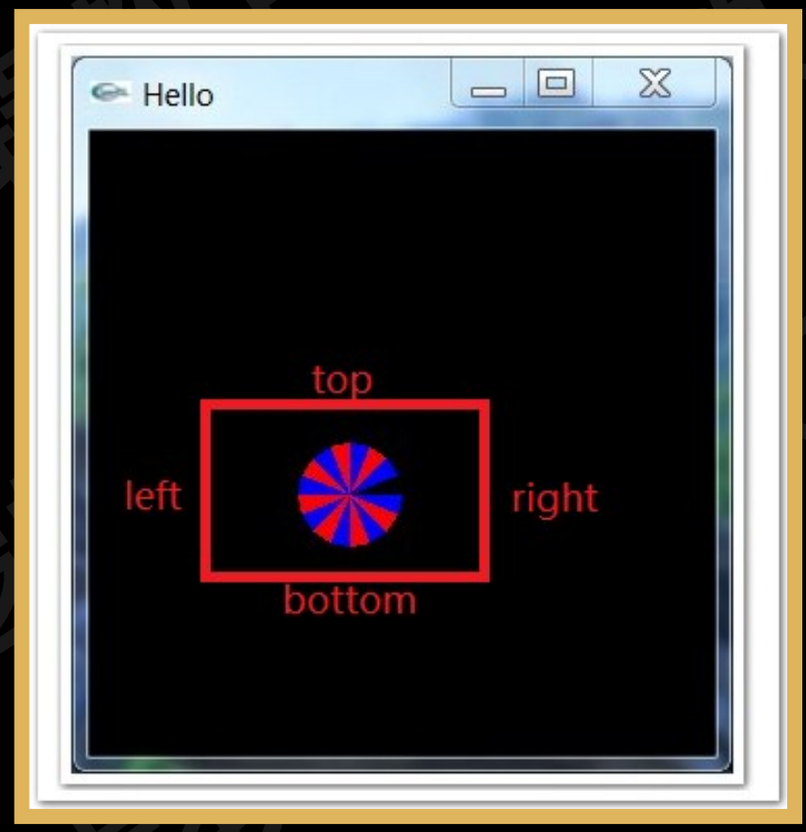
```
//1 开启裁剪测试
glEnable(GL_SCISSOR_TEST);

//2.关闭裁剪测试
glDisable(GL_SCISSOR_TEST);

//3.指定裁剪窗口
 void glScissor(Glint x,Glint y,GLSize width,GLSize height);
x,y:指定裁剪框左下角位置；
width , height:指定裁剪尺寸
```

课程研发:CC老师
课程授课:CC老师

理解窗口,视口,裁剪区域

- 窗口: 就是显示界面
- 视口: 就是窗口中用来显示图形的一块矩形区域，它可以和窗口等大，也可以比窗口大或者小。只有绘制在视口区域中的图形才能被显示，如果图形有一部分超出了视口区域，那么那一部分是看不到的。通过glViewport()函数设置。
- 裁剪区域（平行投影）：就是视口矩形区域的最小最大x坐标（left,right)和最小最大y坐标(bottom,top)，而不是窗口的最小最大x坐标和y坐标。通过glOrtho()函数设置，这个函数还需指定最近最远z坐标，形成一个立体的裁剪区域。





课程研发:CC老师
课程授课:CC老师

逻辑教育
Logic education

课堂案例

实现多层裁剪

课程研发:CC老师
课程授课:CC老师

混合

我们把OpenGL 渲染时会把颜色值存在颜色缓存区中，每个片段的深度值也是放在深度缓冲区。当深度缓冲区被关闭时，新的颜色将简单的覆盖原来颜色缓存区存在的颜色值，当深度缓冲区再次打开时，新的颜色片段只是当它们比原来的值更接近邻近的裁剪平面才会替换原来的颜色片段。

```
glEnable(GL_BIEND);
```

课程研发:CC老师
课程授课:CC老师

组合颜色

目标颜色：已经存储在颜色缓存区的颜色值
源颜色：作为当前渲染命令结果进入颜色缓存区的颜色值

当混合功能被启动时，源颜色和目标颜色的组合方式是混合方程式控制的。在默认情况下，混合方程式如下所示：

Cf = (Cs * S) + (Cd * D)

Cf： 最终计算参数的颜色
Cs： 源颜色
Cd： 目标颜色
S：源混合因子
D：目标混合因子

课程研发:CC老师
课程授课:CC老师

设置混合因子

设置混合因子，需要用到glBlendFun函数
glBlendFunc(GLenum S,GLenum D);

S:源混合因子
D：目标混合因子

OpenGL 混合因子

函数	RGB混合因子	Alpha混合因子
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(Rs,Gs,Bs)	As
GL_ONE_MINUS_SRC_COLOR	(1,1,1) - (Rs,Gs,Bs)	1-As
GL_DST_COLOR	(Rd,Gd,Bd)	Ad
GL_ONE_MINUS_DST_COLOR	(1,1,1)-(Rd,Gd,Bd)	1-Ad
GL_SRC_ALPHA	(As,As,As)	As
GL_ONE_MINUS_SRC_ALPHA	(1,1,1)-(As,As,As)	1-As
GL_DST_ALPHA	(Ad,Ad,Ad)	Ad
GL_ONE_MINUS_DST_ALPHA	(1,1,1)-(Ad,Ad,Ad)	1-Ad
GL_CONSTANT_COLOR	(Rc,Gc,Bc)	Ac
Gl_ONE_MINUS_CONSTANT_COLOR	(1,1,1)-(Rc,Gc,Bc)	1-Ac
GL_CONSTANT_ALPHA	(Ac,Ac,Ac)	Ac
GL_ONE_MINUS_CONSTANT_COLOR	(1,1,1)-(Ac,Ac,Ac)	1-Ac
GL_SRC_ALPHA_SATURATE	(f,f,f)* f = min(As,1-Ad)	1

表中R、G、B、A 分别代表 红、绿、蓝、alpha。
表中下标S、D，分别代表源、目标
表中C 代表常量颜色（默认黑色）

课程研发:CC老师
课程授课:CC老师

课堂案例

下面通过一个常见的混合函数组合来说明问题：

glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

如果颜色缓存区已经有一种颜色红色（1.0f,0.0f,0.0f,0.0f）,这个目标颜色Cd，如果在这上面用一种alpha为0.6的蓝色（0.0f,0.0f,1.0f,0.6f）

Cd (目标颜色) = （1.0f,0.0f,0.0f,0.0f）；
Cs (源颜色) = （0.0f,0.0f,1.0f,0.6f）；
S = 源alpha值 = 0.6f
D = 1 - 源alpha值= 1-0.6f = 0.4f

方程式Cf = (Cs * S) + (Cd * D)

等价于 = （Blue * 0.6f） + (Red * 0.4f)

课程研发:CC老师
课程授课:CC老师

总结

最终颜色是以原先的红色（目标颜色）与 后来的蓝色（源颜色）进行组合。源颜色的alpha值越高，添加的蓝色颜色成分越高，目标颜色所保留的成分就会越少。
混合函数经常用于实现在其他一些不透明的物体前面绘制一个透明物体的效果。

课程研发:CC老师
课程授课:CC老师

逻辑教育 Logic education

课堂案例

在第一次课程的基础上，实现移动矩形方块，颜色重叠处理！

课程研发:CC老师
课程授课:CC老师

改变组合方程式

默认混合方程式：

Cf = (Cs * S) + (Cd * D)

实际上远不止这一种混合方程式，我们可以从5个不同的方程式中进行选择

选择混合方程式的函数：

glbBlendEquation(GLenum mode);

可用的混合方程模式

模式	函数
GL_FUNC_ADD	Cf =(Cs * S) + (Cd * D)
GL_FUNS_SUBTRACT	Cf = (Cs * S) – (Cd * D)
GL_FUNC_REVERSE_SUBTRACT	Cf = (Cd * D) – (Cs * S)
GL_MIN	Cf = min(Cs ,Cd)
GL_MAX	Cf = min(Cs ,Cd)

课程研发:CC老师
课程授课:CC老师

glBlendFuncSeparate 函数

除了能使用glBlendFunc 来设置混合因子，还可以有更灵活的选择。

void glBlendFuncSeparate(GLenum strRGB,GLenum dstRGB ,GLenum strAlpha,GLenum dstAlpha);

strRGB: 源颜色的混合因子
dstRGB: 目标颜色的混合因子
strAlpha: 源颜色的Alpha因子
dstAlpha: 目标颜色的Alpha因子

课程研发:CC老师
课程授课:CC老师

glBlendFuncSeparate 注意

- glBlendFunc 指定 源和目标 RGBA值的混合函数；但是glBlendFuncSeparate函数则允许为RGB 和 Alpha 成分单独指定混合函数。
- 在混合因子表中，GL_CONSTANT_COLOR,GL_ONE_MINUS_CONSTANT_COLOR,GL_CONSTANT_ALPHA,GL_ONE_MINUS_CONSTANT值允许混合方程式中引入一个常量混合颜色。

课程研发:CC老师
课程授课:CC老师

常量混合颜色

常量混合颜色，默认初始化为黑色（0.0f,0.0f,0.0f,0.0f），但是还是可以修改这个常量混合颜色。

```
void glBlendColor(GLclampf red ,GLclampf green ,GLclampf blue ,GLclampf alpha );
```