8. 纹理和采样其对象

该节大部分属于opengl Compatibility 规范

8.4 像素矩形

颜色、深度和某些其他值的矩形可以使用TexImage*D（参见第8.5节）指定给GL，或者可以使用DrawPixels命令（在第18.1节中描述）转换为片段。这些命令操作的一些参数和操作由CopyPixels（用于从一个帧缓冲区位置复制像素到另一个位置）和ReadPixels（用于从帧缓冲区获取像素值）共享；然而，关于CopyPixels和ReadPixels的讨论将在第9章详细讨论帧缓冲区之后进行。尽管如此，我们在本节中指出，与这些命令相关的参数和状态也与CopyPixels或ReadPixels相关。

有许多参数控制缓冲区对象或客户端内存中像素的编码（用于读写），以及像素在放置到帧缓冲区之前（用于读写和复制）或在从帧缓冲区读取之后的处理方式。这些参数使用PixelStore，*PixelTransfer*和PixelMap*命令设置。

8.4.1 像素存储模式和像素缓冲对象

像素存储模式影响当TexImage*D、TexSubImage*D、CompressedTexImage*D、CompressedTexSubImage*D、DrawPixels和ReadPixels（以及其他命令；参见第14.6.2和14.8节）中的一个命令被执行时的操作。这可能与命令执行的时间不同，如果将命令放置在显示列表中（参见第21.4节）。像素存储模式使用以下命令设置：

```
void PixelStore{if}( enum pname, T param );
```

请注意，这是一个通用形式，实际上应该替换`{if}`为特定的维度，例如`i`或`f`。

`pname`是一个表示要设置的参数的符号常量，而`param`是要将其设置为的值。表8.1和18.1总结了像素存储参数、它们的类型、初始值以及允许的范围。

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	[0, ∞)
UNPACK_SKIP_ROWS	integer	0	[0, ∞)
UNPACK_SKIP_PIXELS	integer	0	[0, ∞)
UNPACK_ALIGNMENT	integer	4	1, 2, 4, 8
UNPACK_IMAGE_HEIGHT	integer	0	[0, ∞)
UNPACK_SKIP_IMAGES	integer	0	[0, ∞)
UNPACK_COMPRESSED_BLOCK_WIDTH	integer	0	[0, ∞)
UNPACK_COMPRESSED_BLOCK_HEIGHT	integer	0	[0, ∞)
UNPACK_COMPRESSED_BLOCK_DEPTH	integer	0	[0, ∞)
UNPACK_COMPRESSED_BLOCK_SIZE	integer	0	[0, ∞)

Table 8.1: PixelStore* parameters pertaining to one or more of DrawPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, PolygonStipple, TexImage*D, TexSubImage*D, CompressedTexImage*D and CompressedTexSubImage*D.

数据转换按照第2.2.1节中指定的规则执行。

除了在客户端内存中存储像素数据外，像素数据也可以存储在缓冲区对象中（在第6节中描述）。当前的像素解包和打包缓冲区对象分别由PIXEL_UNPACK_BUFFER和PIXEL_PACK_BUFFER目标指定。

最初，对于PIXEL_UNPACK_BUFFER，绑定为零，表示图像规范命令（例如DrawPixels）的像素来自客户端内存指针参数。然而，如果绑定了一个非零缓冲区对象作为当前像素解包缓冲区，那么指针参数将被视为指向指定缓冲区对象的偏移量。

8.4.2 影像子集

一些像素传输操作仅在GL实现中提供，这些实现包含可选的影像子集。影像子集包括新命令和作为现有命令参数的新枚举值。如果支持该子集，则必须按照本节稍后描述的方式实现所有这些调用和枚举值。

仅在影像子集中提供的个别操作在第8.4.3节中描述。影像子集操作包括：
• 调色板，包括在子节调色板规范、备用调色板规范命令、调色板状态和代理状态、调色板查找、后卷积调色板查找和后颜色矩阵调色板查找中描述的所有命令和枚举值，以及第8.4.3.4节中描述的查询命令。
• 卷积，包括在子节卷积滤波器规范、备用卷积滤波器规范命令和卷积中描述的所有命令和枚举值，以及第8.4.3.8节中描述的查询命令。
• 色彩矩阵，包括在子节颜色矩阵规范和颜色矩阵变换中描述的所有命令和枚举值，以及第8.4.3.11节中描述的简单查询命令。
• 直方图和最小最大值，包括在子节直方图表规范、直方图状态和代理状态、直方图、最小最大值表规范和最小最大值中描述的所有命令和枚举值，以及第8.4.3.13节和第8.4.3.16节中描述的查询命令。
只有在EXTENSIONS字符串包含子字符串"GL_ARB_imaging"时才支持影像子集。查询EXTENSIONS的方法在第22.2节中描述。

如果不支持影像子集，相关的像素传输操作将不会执行；像素将不经修改地传递到下一个操作。

8.4.3 像素传输模式

像素传输模式影响DrawPixels（第18.1节）、ReadPixels（第18.2节）和CopyPixels（第18.3节）在执行其中一个命令时的操作（这可能与发出命令的时间不同）。一些像素传输模式使用以下命令设置：

```
void PixelTransfer{if}( enum param, T value );
```

`param`是一个表示要设置的参数的符号常量，而`value`是要将其设置为的值。表8.2总结了使用PixelTransfer设置的像素传输参数，它们的类型、初始值以及允许的范围。

数据转换按照第2.2.1节中指定的规则执行。

如果`value`超出表8.2中`param`的给定范围，将生成INVALID_VALUE错误。

像素映射查找表使用以下命令设置：

```
void PixelMap{ui us f}v( enum map, sizei size, const T *values );
```

`map`是一个表示要设置的映射的符号映射名称，`size`指示映射的大小，`values`是一个大小为`map`值的数组。

表的条目可以使用三种类型之一指定：单精度浮点、无符号短整数或无符号整数，具体取决于调用的PixelMap的三个版本中的哪一个。当指定表条目时，将其转换为适当的类型。给出颜色分量值的条目按照方程式2.1描述的方式转换，然后被夹紧到范围[0, 1]。给出颜色索引值的条目从无符号短整数或无符号整数转换为浮点数。给出模板索引的条目从单精度浮点转换为最近的整数。各种表及其初始大小和条目在表8.3中总结。

每个表的最大允许大小由MAX_PIXEL_MAP_TABLE的依赖于实现的值指定，但必须至少为32（所有表都应用于单个最大值）。

如果绑定了像素解包缓冲区（由PIXEL_UNPACK_BUFFER_BINDING的非零值指示），则`values`是像素解包缓冲区中的偏移量；否则，`values`是指向客户端内存的指针。在指定像素映射时，将忽略所有像素存储和像素传输模式。读取n个机器单位，其中n是像素映射的大小乘以浮点数、uint或ushort数据在基本机器单位中的大小，具体取决于相应的PixelMap版本。

8.4.3.1 像素映射查询

以下命令：

```
void GetPixelMap{ui us f}v( enum map, T *data );
void GetnPixelMap{ui us f}v( enum map, sizei bufSize, T data );
```

在`data`中返回像素映射`map`中的所有值。`map`必须是表8.3中的映射名称。Get*PixelMapuiv*和*Get*PixelMapusv将浮点像素映射值根据表18.2中的UNSIGNED_INT和UNSIGNED_SHORT条目转换为整数。

GetnPixelMap*不会写入超过`bufSize`字节到`data`中。

如果绑定了像素打包缓冲区（由PIXEL_PACK_BUFFER_BINDING的非零值指示），则`data`是像素打包缓冲区中的偏移量；否则，`data`是指向客户端内存的指针。在返回像素映射时，将忽略所有像素存储和像素传输模式。写入n个机器单位，其中n是像素映射的大小分别乘以FLOAT、UNSIGNED_INT或UNSIGNED_SHORT的基本机器单位中的大小。

8.4.3.2 色彩表规范

使用以下命令指定颜色查找表：

```
void ColorTable( enum target, enum internalformat, sizei width, enum format, enum
type, const void *data );
```

`target`必须是表8.4中列出的常规颜色表名称之一，以定义表。代理表名称是稍后在本节讨论的一种特殊情况。`width`、`format`、`type`和`data`指定了内存中的一个图像，其含义和允许的值与DrawPixels（见第18.1节）的相应参数相同，其中`height`被视为1。表的最大允许宽度是依赖于实现的，但必须至少为32。不允许使用格式COLOR_INDEX、DEPTH_COMPONENT、DEPTH_STENCIL和STENCIL_INDEX以及类型BITMAP。

指定的图像从内存中取出并处理，就像调用DrawPixels一样，在最终扩展到RGBA之后停止。然后，每个像素的R、G、B和A分量由四个COLOR_TABLE_SCALE参数进行缩放，并由四个COLOR_TABLE_BIAS参数进行偏移。可以通过调用ColorTableParameterfv设置这些参数，如下所述。如果启用了片段颜色夹紧或`internalformat`是定点的，那么分量将夹紧到[0, 1]。否则，分量不会被修改。

然后从得到的R、G、B和A值中选择分量，以获得一个具有由`internalformat`指定（或派生自`internalformat`）的基本内部格式的表，与纹理的方式相同（第8.5节）。`internalformat`必须是表8.18或表8.19-8.21中的格式之一，除了在这些表中的RED、RG、DEPTH_COMPONENT和DEPTH_STENCIL基本和大小内部格式之外，所有带有非固定内部数据类型的大小内部格式（参见第8

节）以及大小内部格式RGB9_E5。

颜色查找表被重新定义为具有`width`个条目，每个具有指定的内部格式。该表通过索引0到`width-1`进行形成。表位置`i`由第`i`个图像像素指定，从零开始计数。

表的缩放和偏移参数通过调用以下命令指定：

```
void ColorTableParameter{if}v( enum target, enum pname, const T *params );
```

`target`必须是常规颜色表名称。`pname`是COLOR_TABLE_SCALE或COLOR_TABLE_BIAS之一。`params`指向包含四个值的数组：红色、绿色、蓝色和 alpha，按照这个顺序。

数据转换按照第2.2.1节中指定的规则执行。

GL实现可以根据任何ColorTable参数变化其内部分量分辨率的分配，但是分配不能是任何其他因素的函数，并且一旦确定就不能更改。分配必须是不变的；每次使用相同的参数值指定颜色表时都必须进行相同的分配。这些分配规则也适用于代理颜色表，稍后在本节中描述。

8.4.3.3 备用颜色表规范命令

颜色表也可以使用直接从帧缓冲区获取的图像数据指定，并且可以重新指定现有表的部分。
以下命令：

```
void CopyColorTable( enum target, enum internalformat, int x, int y, sizei width
);
```

以与ColorTable完全相同的方式定义颜色表，只是表数据来自帧缓冲区而不是客户端内存。`target`必须是常规颜色表名称。`x`、`y`和`width`与CopyPixels的相应参数完全对应（参见第18.3节）；它们指定要复制的帧缓冲区区域的宽度以及左下角（x，y）坐标。图像从帧缓冲区中获取，就像将这些参数传递给CopyPixels时，参数类型设置为COLOR，高度设置为1一样，在最终扩展到RGBA之后停止。

随后的处理与ColorTable所描述的完全相同，从COLOR_TABLE_SCALE开始。参数`target`、`internalformat`和`width`使用与ColorTable的相应参数相同的值指定，并具有相同的含义。`format`被视为RGBA。

还有两个额外的命令：

```
void ColorSubTable( enum target, sizei start, sizei count, enum format, enum
type, const void *data );
void CopyColorSubTable( enum target, sizei start, int x, int y, sizei count );
```

它们重新指定现有颜色表的部分。不会更改指定颜色表的internalformat或width参数，也不会更改指定部分以外的表条目。`target`必须是常规颜色表名称。

`ColorSubTable`的参数`format`、`type`和`data`与ColorTable的相应参数匹配，这意味着它们使用相同的值进行指定，并具有相同的含义。同样，`CopyColorSubTable`的参数`x`、`y`和`count`与`CopyColorTable`的`x`、`y`和`width`参数匹配。`ColorSubTable`命令和`CopyColorSubTable`命令都以与它们的`ColorTable`对应项完全相同的方式解释和处理像素组，只是将R、G、B和A像素组值分配给颜色表组件的方式由表的`internalformat`控制，而不是由命令的参数控制。

`ColorSubTable`和`CopyColorSubTable`的`start`和`count`参数指定颜色表的一个子区域，从索引`start`开始，到索引`start + count - 1`结束。从零开始计数，第`n`个像素组被分配给具有索引`count + n`的表条目。

8.4.3.4 颜色表查询

要查询颜色表的当前内容，可以使用以下命令：

颜色表的当前内容可以使用以下命令查询：

```
void GetColorTable( enum target, enum format, enum type, void *table );
void GetnColorTable( enum target, enum format, enum type, sizei bufSize, void
*table );
```

`target`必须是表8.4中列出的常规颜色表名称之一。`format`必须是表8.5中的像素格式，`type`必须是表8.6中的数据类型。一维颜色表图像将返回到像素打包缓冲区或客户端内存，起始位置为`table`。在这个图像上不执行任何像素传输操作，但会执行适用于ReadPixels的像素存储模式。在指定格式中请求的颜色分量，但不包括在颜色查找表的内部格式中的分量，将返回为零。将内部颜色分量分配给格式请求的分量的分配方式在表8.26中描述。

以下命令用于整数和浮点数查询：

```
void GetColorTableParameter{if}v( enum target, enum pname, T params );
```

`target`必须是表8.4中列出的常规或代理颜色表名称之一。`pname`必须是COLOR_TABLE_SCALE、COLOR_TABLE_BIAS、COLOR_TABLE_FORMAT、COLOR_TABLE_WIDTH、COLOR_TABLE_RED_SIZE、COLOR_TABLE_GREEN_SIZE、COLOR_TABLE_BLUE_SIZE、COLOR_TABLE_ALPHA_SIZE、COLOR_TABLE_LUMINANCE_SIZE或COLOR_TABLE_INTENSITY_SIZE之一。指定参数的值将在`params`中返回。

8.4.3.5 色彩表状态和代理状态

与色彩表相关的状态可以分为两类。对于三种表中的每一种，都有一个数值数组。每个数组都有一个关联的宽度，一个整数描述表的内部格式，六个整数值描述表的红色、绿色、蓝色、alpha、亮度和强度分量的分辨率，以及两组四个浮点数来存储表的缩放和偏移。每个初始数组都为空（零宽度，内部格式为RGBA，带有零大小的分量）。缩放参数的初始值为（1,1,1,1），偏移参数的初始值为（0,0,0,0）。

除了颜色查找表之外，还维护了部分实例化的代理颜色查找表。每个代理表都包括宽度和内部格式状态值，以及红色、绿色、蓝色、alpha、亮度和强度分辨率的状态。代理表不包括图像数据，也不包括缩放和偏移参数。当使用在表8.4中列出的代理颜色表名称之一指定目标执行ColorTable时，表的代理状态值将被重新计算和更新。如果表太大，不会生成错误，但代理格式、宽度和分量分辨率将被设置为零。如果使用将目标设置为相应的常规表名称（例如，COLOR_TABLE是PROXY_COLOR_TABLE的对应常规名称）调用ColorTable将容纳颜色表，则代理状态值将被设置，就好像正在指定常规表一样。使用代理目标调用ColorTable对任何实际颜色表的图像或状态没有影响。

对于任何代理目标都没有与之关联的图像。它们不能用作颜色表，也不能使用GetColorTable进行查询。

8.4.3.6 卷积滤波器规范

通过调用以下命令来指定二维卷积滤波器图像：

```
void ConvolutionFilter2D( enum target, enum internalformat, sizei width, sizei
height, enum format, enum type, const void *data );
```

`target`必须是CONVOLUTION_2D。`width`、`height`、`format`、`type`和`data`指定了内存中的一幅图像，其含义和允许的值与相应的DrawPixels参数相同。不允许使用格式COLOR_INDEX、DEPTH_COMPONENT、DEPTH_STENCIL和STENCIL_INDEX以及类型BITMAP。指定的图像从内存中提取出来，并且处理方式与调用DrawPixels一样，在最终扩展到RGBA之后停止。然后，每个像素的R、G、B和A分量由四个二维CONVOLUTION_FILTER_SCALE参数进行缩放，并由四个二维CONVOLUTION_FILTER_BIAS参数进行偏置。通过调用ConvolutionParameterfv来设置这些参数，如下所述。在此过程中，在任何时候都不进行夹持。
然后，从生成的R、G、B和A值中选择组件，以获取由`internalformat`指定的或从中派生的基本内部格式的表，与纹理相同的方式（第8.5节）。`internalformat`接受与ColorTable的相应参数相同的值。
像素的红、绿、蓝、alpha、亮度和/或强度分量以浮点格式存储，而不是整数格式。它们形成一个二维图像，由坐标i、j索引，其中i从左到右递增，从零开始，j从下到上递增，同样从零开始。图像位置i、j由第N个像素指定，从零开始计数，其中N = i + j * width。

- 二维滤波器的缩放和偏移参数由以下命令指定：

```
void ConvolutionParameter{if}v( enum target, enum pname, const T *params );
```

`target`为CONVOLUTION_2D，`pname`为CONVOLUTION_FILTER_SCALE或CONVOLUTION_FILTER_BIAS之一。`params`指向包含四个值的数组：红色、绿色、蓝色和alpha，按照这个顺序。
数据转换按照第2.2.1节中指定的规则执行。

一维卷积滤波器使用以下命令定义：

```
void ConvolutionFilter1D( enum target, enum internalformat, sizei width, enum
format, enum type, const void *data );
```

`target`必须是CONVOLUTION_1D。`internalformat`、`width`、`format`和`type`的语义相同，接受与它们的二维对应项相同的值。然而，`data`必须指向一维图像。
该图像从内存中提取并处理，就好像调用ConvolutionFilter2D并将高度设置为1一样，只是它由一维CONVOLUTION_FILTER_SCALE和CONVOLUTION_FILTER_BIAS参数进行缩放和偏置。这些参数与二维参数完全相同，只是调用ConvolutionParameterfv时`target`为CONVOLUTION_1D。
图像以使i递增从左到右开始，从零开始。图像位置i由第i个像素指定，从零开始计数。

提供了特殊的功能来定义二维可分离滤波器 - 其图像可以表示为两个一维图像的乘积，而不是完整的二维图像。可以使用以下命令指定二维可分离卷积滤波器：

```
void SeparableFilter2D( enum target, enum internalformat, sizei width, sizei
height, enum format, enum type, const void *row, const void *column );
```

`target`必须是SEPARABLE_2D。`internalformat`指定了将保留的两个一维图像的表条目的格式。`row`指向指定格式和类型的宽为width像素的图像。`column`指向指定格式和类型的高为height像素的图像。
这两个图像从内存中提取并处理，就好像为每个图像分别调用ConvolutionFilter1D一样，只是每个图像都会由二维可分离CONVOLUTION_FILTER_SCALE和CONVOLUTION_FILTER_BIAS参数进行缩放和偏置。这些参数与一维和二维参数完全相同，只是在调用ConvolutionParameteriv时`target`为SEPARABLE_2D。

8.4.3.7 备用卷积滤波器规范命令

可以使用直接从帧缓冲区获取的图像数据指定一维和二维滤波器。
以下命令可以定义二维滤波器，方式与ConvolutionFilter2D完全相同，只是图像数据来自帧缓冲而不是客户端内存：

```
void CopyConvolutionFilter2D( enum target, enum internalformat, int x, int y,
sizei width, sizei height );
```

`target`必须是CONVOLUTION_2D。`x`、`y`、`width`和`height`与CopyPixels的相应参数完全对应（参见第18.3节）；它们指定了图像的宽度和高度，以及要复制的帧缓冲区区域的左下角（x，y）坐标。图像从帧缓冲区中提取，处理方式与将这些参数传递给CopyPixels并将参数类型设置为COLOR一样，在最终扩展到RGBA之后停止。
后续处理与ConvolutionFilter2D的描述完全相同，从CONVOLUTION_FILTER_SCALE开始。参数`target`、`internalformat`、`width`和`height`使用相同的值指定，具有相同的含义，与ConvolutionFilter2D的相应参数相同。`format`被视为RGBA。

以下命令定义了一维滤波器，方式与ConvolutionFilter1D完全相同，只是图像数据来自帧缓冲而不是客户端内存：

```
void CopyConvolutionFilter1D( enum target, enum internalformat, int x, int y,
sizei width );
```

`target`必须是CONVOLUTION_1D。`x`、`y`和`width`与CopyPixels的相应参数完全对应（参见第18.3节）；它们指定了图像的宽度以及要复制的帧缓冲区区域的左下角（x，y）坐标。图像从帧缓冲区中提取，处理方式与将这些参数传递给CopyPixels并将参数类型设置为COLOR，高度设置为1一样，在最终扩展到RGBA之后停止。
后续处理与ConvolutionFilter1D的描述完全相同，从CONVOLUTION_FILTER_SCALE开始。参数`target`、`internalformat`和`width`使用相同的值指定，具有相同的含义，与ConvolutionFilter2D的相应参数相同。`format`被视为RGBA。

可以使用以下命令查询卷积滤波器图像的内容：

```
void GetConvolutionFilter( enum target, enum format, enum type, void *image );
void GetnConvolutionFilter( enum target, enum format, enum type, sizei bufSize,
void *image );
```

`target`必须是CONVOLUTION_1D或CONVOLUTION_2D。`format`必须是来自表8.5的像素格式，`type`必须是来自表8.6的数据类型。一维或二维图像被返回到像素打包缓冲区或客户端内存，从`image`开始。像素处理和组件映射与GetTexImage的相同。

可以使用以下命令查询可分离滤波器图像的内容：

```
void GetSeparableFilter( enum target, enum format, enum type, void *row, void
*column, void *span );
void GetnSeparableFilter( enum target, enum format, enum type, sizei rowBufSize,
void *row, sizei columnBufSize, void *column, void *span );
```

`target`必须是SEPARABLE_2D。`format`必须是来自表8.5的像素格式，`type`必须是来自表8.6的数据类型。分别从`row`和`column`开始，将行和列图像返回到像素打包缓冲区或客户端内存。`span`未使用。像素处理和组件映射与GetTexImage的相同。

以下命令用于整数和浮点数查询：

```
void GetConvolutionParameter{if}v( enum target, enum pname, T *params );
```

`target`必须是CONVOLUTION_1D、CONVOLUTION_2D或SEPARABLE_2D。`pname`必须是CONVOLUTION_BORDER_COLOR、CONVOLUTION_BORDER_MODE、CONVOLUTION_FILTER_SCALE、CONVOLUTION_FILTER_BIAS、CONVOLUTION_FORMAT、CONVOLUTION_WIDTH、CONVOLUTION_HEIGHT、MAX_CONVOLUTION_WIDTH或MAX_CONVOLUTION_HEIGHT中的一个。指定参数的值将在`params`中返回。

8.4.3.9 Convolution Filter State

卷积滤波器状态包括一维滤波器图像、两个一维滤波器图像（用于可分离滤波器），以及二维滤波器图像。每个滤波器都具有关联的宽度和高度（仅适用于二维和可分离滤波器），一个整数描述滤波器的内部格式，以及两组四个浮点数，用于存储滤波器的比例和偏移。

每个初始卷积滤波器都是空的（宽度和高度为零，内部格式为RGBA，具有零大小的组件）。所有比例参数的初始值为（1,1,1,1），所有偏移参数的初始值为（0,0,0,0）。

8.4.3.10 Color Matrix Specification

将矩阵模式设置为COLOR会导致在颜色矩阵堆栈上的顶部矩阵上应用在第12.1.1节中描述的矩阵操作。所有矩阵操作对颜色矩阵的影响与它们对其他矩阵的影响相同。

8.4.3.11 Color Matrix Query

使用GetFloatv并将pname设置为适当的变量名称来查询比例和偏移变量。通过调用GetFloatv并将pname设置为COLOR_MATRIX或TRANSPOSE_COLOR_MATRIX，可以返回颜色矩阵堆栈上的顶部矩阵。使用GetIntegerv，将pname设置为COLOR_MATRIX_STACK_DEPTH和MAX_COLOR_MATRIX_STACK_DEPTH，可以查询颜色矩阵堆栈的深度以及颜色矩阵堆栈的最大深度。

8.4.3.12 Histogram Table Specification

直方图表通过以下命令指定：

```
void Histogram( enum target, sizei width, enum internalformat, boolean sink );
```

如果要指定直方图表，则目标必须是HISTOGRAM。目标值PROXY_HISTOGRAM是一个稍后在本节中讨论的特殊情况。width指定直方图表中的条目数，internalformat指定每个表条目的格式。直方图表的最大允许宽度是依赖于实现的，但必须至少为32。sink指定像素组是否将被直方图操作消耗（TRUE）或传递到minmax操作（FALSE）。

指定的直方图表被重新定义为具有width个条目，每个条目具有指定的内部格式。条目从0到width - 1进行索引。每个条目中的每个分量都设置为零。前一个直方图表中的值（如果有）将丢失。

GL实现可以根据任何Histogram参数来变化其内部分量分辨率的分配，但分配不能是任何其他因素的函数，并且一旦确定就不能更改。特别是，分配必须是不变的；每次使用相同的参数值指定直方图时，都必须进行相同的分配。这些分配规则也适用于稍后在本节中描述的代理直方图。

8.4.3.13 直方图查询

使用以下命令查询直方图表的内容：

```
void GetHistogram( enum target, boolean reset, enum format, enum type, void
*values );
void GetnHistogram( enum target, boolean reset, enum format, enum type, sizei
bufSize, void *values );
```

目标必须是HISTOGRAM。格式必须是来自表8.5的像素格式，类型必须是来自表8.6的数据类型。一维直方图表图像返回到像素包装缓冲区或从values开始的客户端内存。像素处理和分量映射与GetTexImage的相同，只是不是应用最终的转换像素存储模式，而是简单地将分量值夹在目标数据类型的范围内。

如果reset为TRUE，则直方图的所有元素的所有计数器都将重置为零。计数器是否返回或不返回都会重置。

如果reset为FALSE，则不修改任何计数器。

8.4.3.14 直方图状态和代理状态

直方图操作所需的状态包括一个值数组，其中关联有一个宽度，一个描述直方图内部格式的整数，五个整数值描述表的每个红色、绿色、蓝色、alpha和亮度分量的分辨率，以及一个指示像素组是否由操作消耗的标志。初始数组为空（零宽度，内部格式RGBA，零大小的分量）。标志的初始值为FALSE。

除了直方图表之外，还维护了一个部分实例化的代理直方图表。它包括宽度、内部格式以及红色、绿色、蓝色、alpha和亮度分量分辨率。代理表不包括图像数据或标志。当使用目标设置为PROXY_HISTOGRAM执行直方图时，重新计算并更新代理状态值。如果直方图数组太大，不会生成错误，但代理格式、宽度和分辨率将被设置为零。如果直方图表可以通过带有目标设置为HISTOGRAM的直方图调用来容纳，代理状态值将被设置为实际直方图表正在被指定的状态。使用目标PROXY_HISTOGRAM调用直方图不会对实际直方图表的图像或状态产生影响。

PROXY_HISTOGRAM没有与之关联的图像。不能将其用作直方图，也不能使用GetHistogram查询其图像。

8.4.3.15 最小-最大表规范

最小-最大表使用以下方式指定：

```
void Minmax( enum target, enum internalformat,
             boolean sink );
```

目标必须是MINMAX。internalformat指定表条目的格式。sink指定像素组是否将被最小-最大操作消耗（TRUE）还是传递给最终转换（FALSE）。

internalformat接受与ColorTable的相应参数相同的值，但不包括值1、2、3和4，以及INTENSITY基本和大小的内部格式。生成的表始终具有2个条目，每个条目的值仅对应于内部格式的分量。

最小-最大操作所需的状态是一个包含两个元素的表（第一个元素存储最小值，第二个元素存储最大值），一个描述表的内部格式的整数，以及一个指示像素组是否由操作消耗的标志。初始状态是将最小表条目设置为最大可表示值，并将最大表条目设置为最小可表示值。内部格式设置为RGBA，标志的初始值为FALSE。

8.4.3.16 最小-最大查询

通过以下命令查询最小-最大表的内容：

```
void GetMinmax( enum target, boolean reset,
                enum format, enum type, void *values );
void GetnMinmax( enum target, boolean reset,
                 enum format, enum type, sizei bufSize, void *values );
```

目标必须是MINMAX。format必须是表8.5中的像素格式，type必须是表8.6中的数据类型。返回一个宽度为2的一维图像，起始位置为values，该图像将传输到像素包装缓冲区或客户端内存。像素处理和分量映射与GetTexImage的相同。

如果reset为TRUE，则将每个最小值重置为最大可表示值，并将每个最大值重置为最小可表示值。无论返回与否，都将重置所有值。

如果reset为FALSE，则不修改任何值。

8.4.3.17 重置最小-最大值及查询参数

通过以下命令重置目标的所有最小和最大值为它们的最大和最小可表示值，其中目标必须是MINMAX：

```
void ResetMinmax( enum target );
```

查询参数使用以下命令进行整数和浮点查询，其中目标必须是MINMAX，pname是MINMAX_FORMAT或MINMAX_SINK：

```
void GetMinmaxParameter{if}v( enum target, enum pname, T *params );
```

指定参数的值将在params中返回。

8.4.4 像素矩形的传输

图8.1展示了在缓冲对象或客户端内存中传输像素的过程。我们按照它们发生的顺序描述此过程的各个阶段。

接受或返回像素矩形的命令包括以下参数（以及特定于它们功能的其他参数）：

- `format` 是一个符号常量，表示内存中的值的含义。
- `width` 和 `height` 分别是要传输的像素矩形的宽度和高度。
- `data` 指的是要绘制的数据。这些数据用`type`指定的 GL 数据类型之一表示。类型令牌值与它们指示的 GL 数据类型之间的对应关系在表8.7中给出。

并非所有 `format` 和 `type` 的组合都是有效的。

有关接受的 `format` 和 `type` 值组合的其他限制将在下面讨论。具体命令可能会加上额外的限制。

8.4.4.1 解包

数据从当前绑定的像素解包缓冲区或客户端内存中作为有符号或无符号字节序列（GL 数据类型 byte 和 ubyte）、有符号或无符号短整数（GL 数据类型 short 和 ushort）、有符号或无符号整数（GL 数据类型 int 和 uint）或浮点值（GL 数据类型 half 和 float）中获取。这些元素根据格式分组成一组一、两、三或四个值，形成一个组。表8.8总结了从内存中获取的组的格式；它还指示了那些生成索引的格式以及生成浮点或整数分量的格式。

如果已绑定像素解包缓冲区（由 PIXEL_UNPACK_BUFFER_BINDING 的非零值表示），则数据是像素解包缓冲区中的偏移量，并且相对于此偏移量从缓冲区解包像素；否则，数据是指向客户端内存的指针，并且相对于指针从客户端内存解包像素。

默认情况下，每种 GL 数据类型的值都按照客户端 GL 绑定语言中的指定方式进行解释。然而，如果启用了 UNPACK_SWAP_BYTES，那么这些值将按照表8.9中的修改位顺序进行解释。修改后的位顺序仅在 GL 数据类型 ubyte 具有八位的情况下定义，并且仅在每个特定的 GL 数据类型具有 8、16 或 32 位表示时才定义。

内存中的组被视为排列成一个矩形。该矩形由一系列行组成，第一行的第一组的第一个元素由数据指向。如果 UNPACK_ROW_LENGTH 的值为零，则一行中的组数为 width；否则，组数为 UNPACK_ROW_LENGTH 的值。如果 p 表示第一行的第一个元素在内存中的位置，则第 N 行的第一个元素由以下公式指示：

$$ p + N \cdot k $$

其中 N 是行号（从零开始计数），k 定义为：

$$ k = \begin{cases} \lceil nl/s \rceil \cdot a/s, & \text{if } a \leq s, \ \lceil n \rceil \cdot l, & \text{if } s < a. \end{cases} $$

其中 n 是组中的元素数量，l 是行中组的数量，a 是 UNPACK_ALIGNMENT 的值，s 是以 GL ubyte 为单位的元素大小。如果每个元素的位数不是 GL ubyte 的位数的 1、2、4 或 8 倍，则对于所有 a 的值，k = nl。

有一种机制可以从较大的包含矩形中选择子矩形的组。该机制依赖于三个整数参数：UNPACK_ROW_LENGTH、UNPACK_SKIP_ROWS 和 UNPACK_SKIP_PIXELS。在从内存中获取第一组之前，数据指针将提前（UNPACK_SKIP_PIXELS）n + （UNPACK_SKIP_ROWS）k 个元素。然后，从内存中获取宽度组的连续元素（不会移动指针），之后指针将提前 k 个元素。以这种方式获取 width 组的值，这样就得到了 height 组 width 组的值。请参见图8.2。

8.4.4.2 特殊解释

与表8.10中的类型匹配的类型是一个特殊情况，其中每个组的所有分量都被打包到一个单独的无符号字节、无符号短整数或无符号整数中，具体取决于类型。如果 `type` 是 `FLOAT_32_UNSIGNED_INT_24_8_REV`，那么每个组的分量包含在两个32位字中；第一个字包含浮点分量，第二个字包含一个打包的24位未使用字段，后跟一个8位索引。每个打包像素的组件数量由类型固定，必须与格式参数中指示的每个组的组件数量相匹配，如表8.10所列。

错误：

- 如果在处理像素矩形的任何命令中发生不匹配，将生成 INVALID_OPERATION 错误。

每个打包像素类型的第一个、第二个、第三个和第四个分量的位字段位置如表8.11-8.14所示。每个位字段都被解释为无符号整数值。
通常，分量被打包，其中第一个分量位于位字段的最高有效位，随后的分量占据逐渐减小的有效位置。类型的标记名称以 `_REV` 结尾的类型会颠倒分量的打包顺序，从最不显著位置到最显著位置。在所有情

况下，每个分量的最高有效位都在位置 i。

8.4.4.3 浮点数转换

此步骤仅适用于浮点数分量组。不适用于索引或整数分量。对于包含组件和索引的组，例如DEPTH_STENCIL，索引不会被转换。
组中的每个元素都被转换为浮点数值。对于无符号或有符号的规范化固定点元素，分别使用方程2.1或2.2。

8.4.4.4 转换为RGB

仅当格式为LUMINANCE或LUMINANCE_ALPHA时应用此步骤。如果格式为LUMINANCE，则将每个包含一个元素的组转换为包含R、G和B（三个）元素的组，通过将原始的单个元素复制到这三个新元素中。如果格式为LUMINANCE_ALPHA，则将每个包含两个元素的组转换为包含R、G、B和A（四个）元素的组，通过将第一个原始元素复制到前三个新元素中，并将第二个原始元素复制到A（第四个）新元素中。

8.4.4.5 最终扩展为RGBA

此步骤仅针对非深度分量组执行。将每个组转换为包含4个元素的组，具体步骤如下：如果组中不包含A元素，则添加A元素并将其设置为整数分量的1或浮点数分量的1.0。如果组中缺少R、G或B中的任何一个元素，则添加每个缺失的元素，并为整数分量赋值为0，浮点数分量赋值为0.0。

8.4.4.6 像素传输操作

此步骤实际上是一系列步骤。由于像素传输操作在绘制、复制和读取像素时（参见第18章）以及在纹理图像的规范化过程中（从内存或帧缓冲区中）等情况下都会等效执行，因此它们在第8.4.5节中单独描述。在完成该部分描述的操作之后，按照以下各节的描述处理组。

8.4.5 像素传输操作
OpenGL定义了六种像素组：

1. 浮点数RGBA分量：每个组包含四个浮点格式的颜色分量：红色、绿色、蓝色和alpha。
2. 整数RGBA分量：每个组包含四个整数格式的颜色分量：红色、绿色、蓝色和alpha。
3. 深度分量：每个组包含一个单一的深度分量。
4. 颜色索引：每个组包含一个单一的颜色索引。
5. 模板索引：每个组包含一个单一的模板索引。
6. 深度/模板：每个组包含一个单一的深度分量和一个单一的模板索引。
   在图像中，对于每个像素组，本节描述的每个操作都会按顺序应用。许多操作仅适用于特定种类的像素组；如果某个操作不适用于给定的组，将被跳过。本节定义的操作不会影响整数RGBA分量像素组。

此步骤仅适用于RGBA分量和深度分量组，以及深度/模板组中的深度分量。每个分量都乘以相应的有符号比例因子：R分量乘以RED_SCALE，G分量乘以GREEN_SCALE，B分量乘以BLUE_SCALE，A分量乘以ALPHA_SCALE，或深度分量乘以DEPTH_SCALE。然后，将结果加到相应的有符号偏移量上：R偏移量是RED_BIAS，G偏移量是GREEN_BIAS，B偏移量是BLUE_BIAS，A偏移量是ALPHA_BIAS，或深度偏移量是DEPTH_BIAS。

8.4.5.1 指数上的算术运算

此步骤仅适用于颜色索引和模板索引组，以及深度/模板组中的模板索引。如果索引是浮点值，则将其转换为定点值，小数点右侧的位数未指定，且至少有dlog2(MAX_PIXEL_MAP_TABLE)e位在小数点左侧。已经是整数的索引保持不变；结果固定点值中的任何分数位都为零。
然后，将固定点索引向左或向右移动|INDEX_SHIFT|位，如果INDEX_SHIFT > 0，则向左移，否则向右

移。在任何情况下，移位都是用零填充的。
然后，将有符号整数偏移INDEX_OFFSET添加到索引中。

8.4.5.2 RGBA到RGBA查找

此步骤仅适用于RGBA分量组，如果MAP_COLOR为FALSE，则跳过此步骤。首先，将每个分量夹紧到范围[0, 1]。与每个R、G、B和A分量元素相关联的都有一个表：用于R的PIXEL_MAP_R_TO_R，用于G的PIXEL_MAP_G_TO_G，用于B的PIXEL_MAP_B_TO_B和用于A的PIXEL_MAP_A_TO_A。每个元素都乘以相应表的大小减1的整数，对于每个元素，通过将此值四舍五入到最接近的整数来找到一个地址。对于每个元素，相应表中的地址值替换该元素。

8.4.5.3 颜色索引查找

此步骤仅适用于颜色索引组。如果调用像素传输操作的GL命令要求生成RGBA分量像素组，则在此步骤执行转换。如果需要RGBA分量像素组，则：
• 将要栅格化组，并且GL处于RGBA模式，或者
• 将组作为图像加载到纹理内存中，或者
• 将组以COLOR_INDEX以外的格式返回到客户端内存。
如果需要RGBA分量组，则使用索引的整数部分来引用4个颜色分量表：PIXEL_MAP_I_TO_R，PIXEL_MAP_I_TO_G，PIXEL_MAP_I_TO_B和PIXEL_MAP_I_TO_A。每个表都必须有2^n个条目，其中n是某个整数值（对于每个表，n可能不同）。对于每个表，索引首先四舍五入到最接近的整数；结果与2^n - 1进行与运算，并将得到的值用作表中的地址。
索引的值变为R、G、B或A值，视情况而定。这样获得的四个元素组替换索引，将组的类型更改为RGBA分量。

如果不需要RGBA分量组，并且启用了MAP_COLOR，则在PIXEL_MAP_I_TO_I表中查找索引（否则，不查找索引）。同样，该表必须具有某个整数n的2^n个条目。索引首先四舍五入到最接近的整数；结果与2^n - 1进行与运算，并将得到的值用作表中的地址。表中的值替换索引。浮点表值首先四舍五入为带有未指定精度的定点值。组的类型保持为颜色索引。

8.4.5.4 模板索引查找

此步骤仅适用于模板索引组，以及深度/模板组中的模板索引。如果启用了MAP_STENCIL，则在PIXEL_MAP_S_TO_S表中查找索引（否则，不查找索引）。该表必须具有某个整数n的2^n个条目。整数索引与2^n - 1进行与运算，并将得到的值用作表中的地址。表中的整数值替换索引。

8.4.5.5 颜色表查找

此步骤仅适用于RGBA分量组。只有在启用COLOR_TABLE时才进行颜色表查找。如果启用了零宽度表，则不执行查找。
表的内部格式决定了组的哪些分量将被替换（请参见表8.16）。将要替换的分量通过夹紧到[0, 1]，乘以表宽度减1的整数，并四舍五入到最接近的整数来转换为索引。分量将被表中索引处的表项替换。
所需的状态是一个指示颜色表查找是启用还是禁用的一位。在初始状态下，查找被禁用。

8.4.5.6 卷积

此步骤仅适用于RGBA分量组。如果启用了CONVOLUTION_1D，则一维卷积滤波器仅应用于传递给TexImage1D、TexSubImage1D、CopyTexImage1D和CopyTexSubImage1D的一维纹理图像。如果启用了CONVOLUTION_2D，则二维卷积滤波器仅应用于传递给DrawPixels、CopyPixels、ReadPixels、TexImage2D、TexSubImage2D、CopyTexImage2D、CopyTexSubImage2D和CopyTexSubImage3D的二维图像。如果启用了SEPARABLE_2D，并且禁用了CONVOLUTION_2D，则对这些图像应用可分离的二维卷积滤波器。

卷积操作是源图像像素和卷积滤波器像素的乘积之和。源图像像素始终有四个分量：红色、绿色、蓝色和Alpha，以下方程中表示为Rs、Gs、Bs和As。滤波器像素可以存储在五种格式之一中，具有1、2、3或4个分量。在以下方程中，这些分量表示为Rf、Gf、Bf、Af、Lf和If。卷积操作的结果是4元组R、G、B、A。根据滤波器的内部格式，每个源图像像素的各个颜色分量与一个滤波器分量卷积，或者保持不变。这在表8.17中定义。
三种卷积滤波器的定义不同。变量Wf和Hf是卷积滤波器的尺寸。变量Ws和Hs是源像素图像的尺寸。卷积方程定义如下，其中C是滤波结果，Cf是一维或二维卷积滤波器，Crow和Ccolumn是构成二维可分离滤波器的两个一维滤波器。C
0
s依赖于源图像颜色Cs和卷积边界模式，如下所述。Cr，过滤后的输出图像，取决于所有这些变量，并且为每种边界模式单独描述。像素索引命名规则在第8.4.3.5节中描述。
一维滤波器：
[C[i_0] = \sum_{n=0}^{W_f-1} C_0^s[i_0 + n] \times C_f[n]]

二维滤波器：
[C[i_0, j_0] = \sum*{n=0}^{W_f-1} \sum*{m=0}^{H_f-1} C_0^s[i_0 + n, j_0 + m] \times C_f[n, m]]

可分离的二维滤波器：
[C[i_0, j_0] = \sum*{n=0}^{W_f-1} \sum*{m=0}^{H_f-1} C_0^s[i_0 + n, j_0 + m] \times Crow[n] \times Ccolumn[m]]

如果一维滤波器的Wf为零，则始终将C[i]设置为零。同样，如果二维滤波器的Wf或Hf为零，则始终将C[i, j]设置为零。
特定卷积滤波器的卷积边界模式由调用以下函数指定：

```
void ConvolutionParameter{if}(GLenum target, GLenum pname, T param);
```

其中target是滤波器的名称，pname是CONVOLUTION_BORDER_MODE，param是REDUCE、CONSTANT_BORDER或REPLICATE_BORDER之一。

8.4.5.7 边界模式 REDUCE

使用边界模式REDUCE卷积的源图像的宽度和高度分别减小了Wf-1和Hf-1。如果此减小会生成宽度和/或高度为零或负的结果图像，则输出将简单地为空，不生成错误。由边界模式REDUCE卷积生成的图像的坐标范围为宽度0到Ws-Wf，高度0到Hs-Hf。在可以由规定无效图像尺寸导致错误的情况下，测试的是这些生成的尺寸，而不是源图像的尺寸（具体示例是TexImage1D和TexImage2D，它们为图像尺寸指定了约束。即使调用TexImage1D或TexImage2D时使用了NULL像素指针，生成的纹理图像的尺寸也是从指定图像的卷积中得出的尺寸）。

当边界模式为REDUCE时，C
0
s等于源图像颜色Cs，Cr等于滤波结果C。对于其余的边界模式，定义Cw = Wf/2和Ch = Hf/2。坐标（Cw，Ch）定义卷积滤波器的中心。

8.4.5.8 边界模式 CONSTANT_BORDER

如果卷积边界模式是CONSTANT_BORDER，则输出图像的尺寸与源图像相同。卷积的结果与如果源图像被具有与当前卷积边界颜色相同颜色的像素包围相同。每当卷积滤波器扩展到源图像的边缘之一时，常量颜色边框像素将用作滤波器的输入。通过调用ConvolutionParameterfv或ConvolutionParameteriv，并将pname设置为CONVOLUTION_BORDER_COLOR，params包含四个值，构成RGBA颜色，将当前卷积边界颜色设置为整数颜色分量线性解释，使得最大正整数映射到1.0，最小负整数映射到-1.0。指定浮点颜色分量时，不会对其进行夹紧。

对于一维滤波器，结果颜色由以下方程定义：
[Cr[i] = C[i - Cw]]
其中C[i
0
]由以下方程计算C
0
s
[i
0
]：
[C_0^s[i_0] = \begin{cases}
Cs[i_0] & 0 \leq i_0 < Ws \
Cc & \text{otherwise}
\end{cases}]
其中Cc是卷积边界颜色。

对于二维或可分离的二维滤波器，结果颜色由以下方程定义：
[Cr[i, j] = C[i - Cw, j - Ch]]
其中C[i
0
, j0
]由以下方程计算C
0
s
[i
0
, j0
]：
[C_0^s[i_0, j0] = \begin{cases}
Cs[i_0, j0] & 0 \leq i_0 < Ws, 0 \leq j_0 < Hs \
Cc & \text{otherwise}
\end{cases}]

8.4.5.9 边界模式 REPLICATE_BORDER

卷积边界模式REPLICATE_BORDER也产生与源图像相同尺寸的输出图像。此模式的行为与CONSTANT_BORDER模式相同，除了对卷积滤波器扩展到源图像边缘之外的像素位置的处理方式。对于这些位置，就好像源图像的最外层单像素边框被复制一样。在概念上，源图像的最左侧单像素列中的每个像素被复制Cw次，以在左侧生成额外的图像数据，源图像最右侧单像素列中的每个像素被复制Cw次，以在右侧生成额外的图像数据，源图像最上方和最下方的每个单像素行中的像素值都被复制，以在顶部和底部生成Ch行图像数据。每个角上的像素值也被复制，以为源图像的每个角的卷积操作提供数据。

对于一维滤波器，结果颜色由以下方程定义：
[Cr[i] = C[i - Cw]]
其中C[i
0
]由以下方程计算C
0
s
[i

0
]：
[C_0^s[i_0] = Cs[\text{clamp}(i_0, Ws)]]
并且夹紧函数clamp(val, max)定义为：
[\text{clamp}(val, max) = \begin{cases}
0 & \text{val < 0} \
val & \text{0} \leq \text{val < max} \
\text{max} - 1 & \text{val} \geq \text{max}
\end{cases}]

对于二维或可分离的二维滤波器，结果颜色由以下方程定义：
[Cr[i, j] = C[i - Cw, j - Ch]]
其中C[i
0
, j0
]由以下方程计算C
0
s
[i
0
, j0
]：
[C_0^s[i_0, j0] = Cs[\text{clamp}(i_0, Ws), \text{clamp}(j_0, Hs)]]

如果执行卷积操作，则生成的图像的每个分量都将通过相应的PixelTransfer参数进行缩放：R分量的POST_CONVOLUTION-
RED_SCALE，G分量的POST_CONVOLUTION_GREEN_SCALE，B分量的POST-
CONVOLUTION_BLUE_SCALE，A分量的POST_CONVOLUTION_ALPHA_SCALE。结果将添加到相应的偏差值：POST_CONVOLUTION_RED_BIAS、POST_CONVOLUTION-
GREEN_BIAS、POST_CONVOLUTION_BLUE_BIAS或POST_CONVOLUTION-
ALPHA_BIAS。所需的状态是三位，指示一维、二维或可分离二维卷积是否启用或禁用，一个整数描述当前的卷积边界模式，以及四个浮点值指定卷积边界颜色。在初始状态下，所有卷积操作都被禁用，边界模式为REDUCE，并且边界颜色为（0, 0, 0, 0）。

8.4.5.10 后卷积颜色表查找

此步骤仅适用于RGBA分量组。后卷积颜色表查找通过调用Enable或Disable，目标为POST_CONVOLUTION_COLOR_TABLE，启用或禁用。后卷积表通过使用目标参数为POST_CONVOLUTION_COLOR_TABLE的ColorTable进行定义。在其他方面，操作与8.4.5.5节中定义的颜色表查找相同。

所需的状态是一个位，指示是否启用或禁用后卷积表查找。在初始状态下，查找被禁用。

8.4.5.11 颜色矩阵变换

此步骤仅适用于RGBA分量组。颜色组件通过颜色矩阵进行变换。每个变换后的分量都乘以相应的有符号比例因子：对于R分量是POST_COLOR_MATRIX_RED_SCALE，对于G分量是POST_COLOR_MATRIX_GREEN_SCALE，对于B分量是POST_COLOR_MATRIX_BLUE_SCALE，对于A分量是POST_COLOR_MATRIX_ALPHA_SCALE。结果再加上有符号的偏差值：对于R分量是POST_COLOR_MATRIX_RED_BIAS，对于G分量是POST_COLOR_MATRIX_GREEN_BIAS，对于B分量是POST_COLOR_MATRIX_BLUE_BIAS，对于A分量是POST_COLOR_MATRIX_ALPHA_BIAS。生成的分量替换原始组的每个分量。

换句话说，如果M_c是颜色矩阵，s的下标表示组件的比例项，b的下标表示偏差项，则组件

[\begin{bmatrix} R \ G \ B \ A \end{bmatrix}]

被变换为

[\begin{bmatrix} R_0 \ G_0 \ B_0 \ A_0 \end{bmatrix} = \begin{bmatrix} R_s & 0 & 0 & 0 \ 0 & G_s & 0 & 0 \ 0 & 0 & B_s & 0 \ 0 & 0 & 0 & A_s \end{bmatrix} \cdot \begin{bmatrix} R \ G \ B \ A \end{bmatrix} + \begin{bmatrix} R_b \ G_b \ B_b \ A_b \end{bmatrix}]

8.4.5.12 后颜色矩阵颜色表查找

此步骤仅适用于RGBA分量组。通过调用Enable或Disable，目标为POST_COLOR_MATRIX_COLOR_TABLE，启用或禁用后颜色矩阵颜色表查找。后颜色矩阵表通过使用目标参数为POST_COLOR_MATRIX_COLOR_TABLE的ColorTable进行定义。在其他方面，操作与8.4.5.5节中定义的颜色表查找相同。

所需的状态是一个位，指示是否启用或禁用后颜色矩阵查找。在初始状态下，查找被禁用。

8.4.5.13 直方图

此步骤仅适用于RGBA分量组。直方图操作通过调用Enable或Disable，目标为HISTOGRAM，启用或禁用。如果表的宽度非零，则从每个像素组的红色、绿色、蓝色和Alpha分量中派生索引R_i、G_i、B_i和A_i（不修改这些分量），通过将每个分量夹紧到[0, 1]，乘以直方图表的宽度减1，并四舍五入到最近的整数。如果直方图表的格式包括红色或亮度，则递增直方图条目R_i的红色或亮度分量1。如果格式包括绿色，则递增直方图条目G_i的绿色分量1。以相同方式递增蓝色和Alpha直方图条目B_i和A_i。如果直方图条目分量递增超过其最大值，则其值变为未定义；这不是错误。

如果Histogram sink参数为FALSE，则直方图操作不会影响正在处理的像素组流。否则，在直方图操作完成后，立即丢弃所有RGBA像素组。由于直方图在minmax之前，不执行minmax操作。不会生成像素片段，不会更改纹理内存内容，也不会返回像素值。然而，无论是否丢弃像素组，都会修改纹理对象状态。

8.4.5.14 Minmax

此步骤仅适用于RGBA分量组。通过调用Enable或Disable，目标为MINMAX，启用或禁用minmax操作。如果minmax表的格式包括红色或亮度，则仅当红色分量小于该分量时，红色分量值替换最小表元素的红色或亮度值。同样，如果格式包括红色或亮度，并且组的红色分量大于最大元素的红色或亮度值，则红色组分量将替换红色或亮度的最大分量。如果表的格式包括绿色，则绿色组分量在小于最小值或大于最大值时有条件地替换绿色的最小和/或最大值。蓝色和Alpha组分量以相似的方式进行测试和替换，如果表格格式包括蓝色和/或alpha。

最小值和最大值组件值的内部类型为浮点，至少具有用于表示颜色的浮点数的相同可表示范围（第2.3.4节）。对于超出可表示范围的组分值，未定义其处理方式。

如果Minmax sink参数为FALSE，则minmax操作不会影响正在处理的像素组流。否则，在minmax操作完成后，立即丢弃所有RGBA像素组。不会生成像素片段，不会更改纹理内存内容，

也不会返回像素值。然而，无论是否丢弃像素组，都会修改纹理对象状态。

绘制、读取和复制像素规范

可以使用`DrawPixels`将像素写入帧缓冲区，并使用`ReadPixels`从帧缓冲区读取像素。`CopyPixels`和`BlitFramebuffer`可用于从帧缓冲区的一个部分复制像素到另一个部分。

18.1 绘制像素

在兼容规范定义

可以使用以下命令绘制像素矩形：

```
void DrawPixels(sizei width, sizei height, enum format, enum type, const void
*data);
```

如果绘制帧缓冲区的选定绘制缓冲区具有整数格式，并且没有激活片段着色器，则光栅化的结果是未定义的。

第一行的第一个元素是指向`DrawPixels`传递的指针指向的ubyte的第一个位（如上定义）。第二行的第一个元素是位置`p + k`处ubyte的第一个位（再次如上定义），其中`k`计算为：

```
k = a * 8a
```

存在一种从`BITMAP`图像中选择元素的子矩形的机制。在从内存获取第一个元素之前，传递给`DrawPixels`的指针在`UNPACK_SKIP_ROWS * k` ubytes上有效地增加。然后，忽略`UNPACK_SKIP_PIXELS`位元，获取后续的`width`位元素，而不会推进ubyte指针，之后ubyte指针增加k个字节。以这种方式获取height组width元素。
像素一旦被传输，`DrawPixels`对像素值执行最终转换（参见18.1.1节），然后将它们转换为片段（参见18.1.2节），这些片段与光栅化生成的片段相同（参见第15和16章）。

18.1.1 最终转换

对于颜色索引，最终转换包括通过二进制点左侧的索引位的位屏蔽，位数为2^n-1，其中n是索引缓冲区中的位数。
对于整数RGBA分量，不执行转换。对于浮点RGBA分量，如果启用了片段颜色夹紧，则将每个元素夹紧到[0, 1]，并根据方程2.3可能转换为定点。如果禁用了片段颜色夹紧，则RGBA分量保持不变。通过调用以下函数来控制片段颜色夹紧：

```
void ClampColor(enum target, enum clamp);
```

其中`target`设置为`CLAMP_FRAGMENT_COLOR`。如果`clamp`为`TRUE`，则启用片段颜色夹紧；如果`clamp`为`FALSE`，则禁用片段颜色夹紧。如果`clamp`为`FIXED_ONLY`，则仅当所有启用的颜色缓冲区具有定点分量时才启用片段颜色夹紧。
如果启用了片段颜色夹紧，则查询纹理边框颜色、纹理环境颜色、雾颜色、alpha测试参考值、混合颜色和RGBA清除颜色将在返回它们之前将相应的状态值夹紧到[0, 1]。此行为与GL的先前版本兼容，当指定这些值时，它们会将这些值夹紧。
对于深度分量，根据深度缓冲区的表示方式处理元素。对于定点深度缓冲区，首先将元素夹紧到范围[0, 1]，然后根据窗口z值（参见13.6.1节）将其转换为定点。当深度缓冲区使用浮点表示时，不需要转换，但是需要夹紧。
模板索引通过2^n-1掩码，其中n是模板缓冲区中的位数。
片段颜色夹紧所需的状态是一个三值整数。片段颜色夹紧的初始值为`FIXED_ONLY`。

18.1.2 转换为片段

将组转换为片段的过程由以下函数控制：

```
void PixelZoom(float zx, float zy);
```

让（xrp，yrp）为当前光栅位置（参见14.7节）。如果当前光栅位置无效，则忽略`DrawPixels`；像素传输操作不更新直方图或minmax表，也不生成片段。然而，即使以后通过像素拥有权测试（见14.9.1节）或剪

刀测试（见14.9.2节）拒绝了相应的片段，直方图和minmax表也会更新。如果特定组（索引或分量）是一行中的第n个，并且属于第m行，则考虑由以下矩形边界的窗口坐标界定的区域，角落为

```
(xrp + zxn, yrp + zym)和(xrp + zx(n + 1), yrp + zy(m + 1))
```

（zx或zy可能为负）。为每个帧缓冲区像素生成表示组（n，m）的片段，该片段在此矩形的内部或底部或左边界上，或者在该矩形的上部或右边界上。
来自包含颜色数据的组的片段采用组的颜色索引或颜色分量以及当前光栅位置的关联深度值，而来自深度分量的片段则采用该组的深度值以及当前光栅位置的关联颜色索引或颜色分量。在这两种情况下，雾坐标取自当前光栅位置的关联光栅距离，次要颜色取自当前光栅位置的关联次要颜色，并且纹理坐标取自当前光栅位置的关联纹理坐标。由带有`DEPTH_STENCIL`或`STENCIL_INDEX`格式的`DrawPixels`生成的组被特殊处理，并在第18.1.4节中描述。

18.1.3 像素矩形多采样光栅化

如果启用了`MULTISAMPLE`，并且`SAMPLE_BUFFERS`的值为1，则使用以下算法对像素矩形进行光栅化。让（Xrp，Yrp）为当前光栅位置。（如果当前光栅位置无效，则忽略`DrawPixels`。）如果特定组（索引或分量）是一行中的第n个，并且属于第m行，则考虑由以下矩形边界的窗口坐标界定的区域，

```
(Xrp + Zx * n, Yrp + Zy * m)
```

和

```
(Xrp + Zx * (n + 1), Yrp + Zy * (m + 1))
```

其中`Zx`和`Zy`是由`PixelZoom`指定的像素缩放因子，可以是正数也可以是负数。为每个帧缓冲区像素生成表示组（n，m）的片段，该片段具有一个或多个位于此矩形的内部，底部或左边界上的采样点。因此产生的每个片段都从该组和当前光栅位置获取其关联数据，其方式与第18.1.2节中的讨论一致。所有深度和颜色采样值被分配相同的值，可以是它们的组（对于深度和颜色组件组）的值，也可以是当前光栅位置的值（如果它们不是）。所有样本值都被分配相同的雾坐标和相同的纹理坐标集，这些坐标来自当前光栅位置。
单个像素矩形将为同一帧缓冲区像素生成多个，可能是非常多的片段，具体取决于像素缩放因子。

18.1.4 写入模板或深度/模板缓冲区

如果`DrawPixels`的`format`参数为`STENCIL_INDEX`或`DEPTH_STENCIL`，则发生`DrawPixels`的所有操作，但窗口坐标（x，y），每个都具有相应的模板索引或深度值和模板索引，产生而不是片段。每个坐标数据对直接发送到每个片段操作，绕过第15和16章中的片段处理操作。最后，每个模板索引写入其指示的帧缓冲区位置，受当前前模板掩码（使用`StencilMask`或`StencilMaskSeparate`设置）的影响。

如果存在深度分量，并且`DepthMask`的设置不为`FALSE`，则还将其写入帧缓冲区；忽略`DepthFunc`的设置。

以上这些内容接口在mesa内部不可用

18.2 读取像素

从帧缓冲区读取像素并将其放入像素包装缓冲区或客户端内存的方法如图18.1所示。我们按照发生的顺序描述像素读取过程的各个阶段。

18.2.1 选择用于读取的缓冲区

当从帧缓冲对象的颜色缓冲区读取像素时，选择用于读取的缓冲区称为读取缓冲区，并由以下命令控制：

```
void ReadBuffer(enum src);
void NamedFramebufferReadBuffer(uint framebuffer, enum src);
```

如果影响了默认帧缓冲区（参见第9节），`src`必须是表17.4中列出的值之一，包括`NONE`。`FRONT_AND_BACK`、`FRONT`和`LEFT`指的是前左缓冲区，`BACK`指的是后左缓冲区，`RIGHT`指的是前右缓冲区。其他常量直接对应它们命名的缓冲区。对于默认帧缓冲区，读取缓冲区的初始值为：如果没有后缓冲区，则为`FRONT`；如果有后缓冲区，则为`BACK`；如果没有与上下文关联的默认帧缓冲区，则为`NONE`。

如果影响了帧缓冲对象，`src`必须是表17.5中列出的值之一，包括`NONE`。指定`COLOR_ATTACHMENTi`会启用从附加到该附件点的帧缓冲区图像中读取。帧缓冲对象的读取缓冲区的初始值为`COLOR_ATTACHMENT0`。

可以通过使用`GetIntegerv`并将`pname`设置为`READ_BUFFER`来查询当前绑定的读取帧缓冲区的读取缓冲区。

18.2.2 ReadPixels

起初，将`PIXEL_PACK_BUFFER`绑定为零，表示图像读取和查询命令（如`ReadPixels`）将像素结果返回到客户端内存指针参数。然而，如果绑定了非零的缓冲对象作为当前像素包装缓冲区，那么指针参数将被视为指定缓冲对象的偏移量。

使用以下命令读取像素：

```
void ReadPixels(int x, int y, GLsizei width, GLsizei height,
                GLenum format, GLenum type, void *data);
void ReadnPixels(int x, int y, GLsizei width, GLsizei height,
                 GLenum format, GLenum type, GLsizei bufSize,
                 void *data);
```

`ReadPixels`中的x和y之后的参数在第8.4.4节中有描述。适用于`ReadPixels`和其他查询图像的命令（参见第8.11节）的像素存储模式总结在表18.1中。

首选的`format`和`type`值可以通过使用`GetIntegerv`并将`pname`分别设置为`IMPLEMENTATION_COLOR_READ_FORMAT`和`IMPLEMENTATION_COLOR_READ_TYPE`来确定。首选的`format`可能会根据当前绑定的读取帧缓冲区的所选读取缓冲区的格式而变化。

18.2.3 从帧缓冲获取像素

- 如果格式为`DEPTH_COMPONENT`，则从深度缓冲获取值。
  - 如果存在多重采样缓冲区（`SAMPLE_BUFFERS`的值为1），则从该缓冲区的深度样本中获取值。建议使用中心样本的深度值，尽管实现可以选择在每个像素处使用深度样本值的任何函数。
- 如果格式为`DEPTH_STENCIL`，则从深度缓冲区和模板缓冲区获取值。`type`必须是`UNSIGNED_INT_24_8`或`FLOAT_32_UNSIGNED_INT_24_8_REV`。
  - 如果存在多重采样缓冲区，则从该缓冲区的深度和模板样本中获取值。建议使用中心样本的深度和模板值，尽管实现可以选择在每个像素处使用深度和模板样本值的任何函数。
- 如果格式为`STENCIL_INDEX`，则从模板缓冲区获取值。
  - 如果存在多重采样缓冲区，则从该缓冲区的模板样本中获取值。建议使用中心样本的模板值，尽管实现可以选择在每个像素处使用模板样本值的任何函数。
- 对于所有其他格式，从读取缓冲区选择的颜色缓冲获取值。

ReadPixels 获取来自每个像素的选择缓冲区的值

- `ReadPixels` 从左下角坐标为 (x+i, y+j) 的每个像素获取值，其中 0 ≤ i < width 且 0 ≤ j < height；这个像素被称为第 j 行的第 i 个像素。如果其中任何像素位于当前GL上下文分配的窗口之外，或者超出当前绑定的读取帧缓冲对象附加的图像之外，那么对于这些像素获取的值是未定义的。当`READ_FRAMEBUFFER_BINDING`为零时，对于不属于当前上下文的单个像素，其值也是未定义的。否则，`ReadPixels`从所选缓冲区获取值，不管这些值是如何放置的。
- 如果`format`是 RED、GREEN、BLUE、RG、RGB、RGBA、BGR 或 BGRA 中的一个，则在每个像素位置从所选缓冲区获取红、绿、蓝和 alpha 值。
- 如果`format`是整数格式，且颜色缓冲区不是整数格式，或者颜色缓冲区是整数格式且`format`不是整数格式，则生成`INVALID_OPERATION`错误。
- 当`READ_FRAMEBUFFER_BINDING`非零时，通过首先读取附加到所选逻辑缓冲区的图像中相应值的内部分量值来获取红、绿、蓝和 alpha 值。通过根据缓冲区的基本内部格式（如表8.11所示）取每个 R、G、B 和 A 分量而将内部分量转换为 RGBA 颜色。如果在内部格式中不存在 G、B 或 A 值，则它们被视为零、零和一，分别。

18.2.4 RGBA值的转换

此步骤仅在`format`不是`STENCIL_INDEX`、`DEPTH_COMPONENT`或`DEPTH_STENCIL`时适用。R、G、B和A值形成一组元素。

- 对于带符号或无符号规格化的定点颜色缓冲区，每个元素都使用方程式2.2或2.1分别转换为浮点数。对于整数或浮点颜色缓冲区，元素保持不变。

18.2.5 深度值的转换

此步骤仅在`format`为`DEPTH_COMPONENT`或`DEPTH_STENCIL`，且深度缓冲使用定点表示时适用。一个元素被视为处于[0, 1]范围内的定点值，其中m是深度缓冲区中的位数（参见第13.6.1节）。如果深度缓冲使用浮点表示，则无需进行转换。

18.2.6 像素传输操作

此步骤实际上是在第8.4.5节中单独描述的步骤序列。在完成该部分描述的处理之后，将按照以下各节的描述处理组。

18.2.7 转换为L

此步骤仅适用于RGBA分量组。如果格式是LUMINANCE或LUMINANCE_ALPHA，则计算值L如下：
[L = R + G + B]
其中R、G和B分别是R、G和B分量的值。计算得到的单个L分量替换了组中的R、G和B分量。

18.2.8 最终转换

读取颜色夹紧通过调用以下函数进行控制：

```
void ClampColor(enum target, enum clamp);
```

其中 `target` 设置为 `CLAMP_READ_COLOR`。如果 `clamp` 为 `TRUE`，则启用读取颜色夹紧；如果 `clamp` 为 `FALSE`，则禁用读取颜色夹紧。如果 `clamp` 为 `FIXED_ONLY`，则仅在所选的读取颜色缓冲区具有固定点分量时启用读取颜色夹紧。

- 对于整数RGBA颜色，每个分量都夹紧到`type`的可表示范围内。
- 对于浮点RGBA颜色，如果 `type` 为 `FLOAT` 或 `HALF_FLOAT`，则如果启用了读取颜色夹紧，则每个分量都夹紧到 [0, 1]。然后，应用来自表18.2的适当转换公式到分量。

如果 `type` 为 `UNSIGNED_INT_10F_11F_11F_REV` 并且 `format` 为 `RGB`，则如果启用了读取颜色夹紧，则每个分量都夹紧到 [0, 1]。然后，返回的数据被打包成一系列 `uint` 值。红色、绿色和蓝色分量分别转换为无符号11位浮点数、无符号11位浮点数和无符号10位浮点数，如第2.3.4.3节和第2.3.4.4节所述。然后，得到的红色11位、绿色11位和蓝色10位被打包为 `UNSIGNED_INT_10F_11F_11F_REV` 格式的第1、2和3个分量，如表8.8所示。

如果 `type` 为 `UNSIGNED_INT_5_9_9_9_REV` 并且 `format` 为 `RGB`，则如果启用了读取颜色夹紧，则每个分量都夹紧到 [0, 1]。然后，返回的数据被打包成一系列 `uint` 值。红色、绿色和蓝色分量在内部格式为RGB9_E5时分别转换为 reds、greens、blues 和 exps 整数，如第8.5.2节所述。然后，reds、greens、blues 和 exps 被打包为 `UNSIGNED_INT_5_9_9_9_REV` 格式的第1、2、3和4个分量，如表8.8所示。

对于其他类型，并且对于浮点或无符号规格化的定点颜色缓冲区，每个分量都会被夹紧到 [0, 1]，无论是否启用了读取颜色夹紧。对于带符号规格化的定点颜色缓冲区，如果启用了读取颜色夹紧，或者 `type` 代表无符号整数分量，则每个分量都夹紧到 [0, 1]；否则，`type` 代表带符号整数分量，每个分量都夹紧到 [-1, 1]。夹紧后，应用来自表18.2的适当转换公式到分量。

对于索引，如果类型不是 FLOAT 或 HALF_FLOAT，则最终转换包括使用表18.3中给定的值进行索引屏蔽。如果类型是 FLOAT 或 HALF_FLOAT，则将整数索引转换为 GL 浮点或半浮点数据值。

18.2.9 放置在像素包装缓冲区或客户端内存中

如果已绑定像素包装缓冲区（由 PIXEL_PACK_BUFFER_BINDING 的非零值表示），则 `data` 是像素包装缓冲区中的偏移量，像素相对于此偏移量被打包到缓冲区中；否则，`data` 是指向客户端内存块的指针，像素相对于指针被打包到客户端内存中。

如果绑定了像素包装缓冲区对象，并且根据像素包装存储状态打包像素数据将访问超出像素包装缓冲区内存大小的范围，则会生成 `INVALID_OPERATION` 错误。

如果绑定了像素包装缓冲区对象，并且 `data` 不能被存储在内存中的相应GL数据类型（表8.2中的 `type` 参数）所需的基本机器单元数量均匀地除尽，则会生成 `INVALID_OPERATION` 错误。

元素组被放置在内存中，就像将像素矩形传输到GL时从内存中取出的那样。也就是说，第j行的第i个组（对应于第j行中的第i个像素）被放置在内存中，就像在传输像素时从内存中取出第j行中的第i个组一样。请参阅第8.4.4.1节中的解包。唯一的区别是使用以 `PACK_` 开头而不是以 `UNPACK_` 开头的存储模式参数。如果格式为 `RED`、`GREEN` 或 `BLUE`，则仅写入相应的单个元素。同样，如果格式为 `RG`、`RGB` 或 `BGR`，则仅写入相应的两个或三个元素。否则，写入每个组的所有元素。

18.3 复制像素

有几个命令可以在帧缓冲区的不同区域之间复制像素数据（参见第18.3.1节），或在纹理和渲染缓冲区的不同区域之间复制像素数据（参见第18.3.2节）。

对于所有这样的命令，如果源和目标相同，或者是相同底层纹理图像的不同视图，并且如果源和目标区域在帧缓冲区、渲染缓冲区或纹理图像中重叠，那么由于复制操作而产生的像素值是未定义的。

18.3.1 复制像素矩形

为了从源帧缓冲区的一个区域传输像素值的矩形到目标帧缓冲区的另一个区域，使用以下命令：

```
void BlitFramebuffer(int srcX0, int srcY0, int srcX1, int srcY1,
                     int dstX0, int dstY0, int dstX1, int dstY1,
                     GLbitfield mask, GLenum filter);

void BlitNamedFramebuffer(uint readFramebuffer, uint drawFramebuffer,
                          int srcX0, int srcY0, int srcX1, int srcY1,
                          int dstX0, int dstY0, int dstX1, int dstY1,
                          GLbitfield mask, GLenum filter);
```

对于 `BlitFramebuffer`，源和目标帧缓冲区分别是绑定到 `READ_FRAMEBUFFER` 和 `DRAW_FRAMEBUFFER` 的帧缓冲区。对于 `BlitNamedFramebuffer`，`readFramebuffer` 和 `drawFramebuffer` 分别是源和目标帧缓冲区对象的名称。

如果未绑定任何帧缓冲区到 `READ_FRAMEBUFFER` 或 `DRAW_FRAMEBUFFER`（对于 `BlitFramebuffer`），或者 `readFramebuffer` 或 `drawFramebuffer` 为零（对于 `BlitNamedFramebuffer`），则默认的读取或绘制帧缓冲区将被用作相应的源或目标帧缓冲区。

`mask` 为零或按位 OR 运算的一个或多个值，指示要复制哪些缓冲区。这些值是 `COLOR_BUFFER_BIT`、`DEPTH_BUFFER_BIT` 和 `STENCIL_BUFFER_BIT`，这些在第17.4.3节中有描述。与这些缓冲区对应的像素从由位置（`srcX0`，`srcY0`）和（`srcX1`，`srcY1`）限定的源矩形复制到由位置（`dstX0`，`dstY0`）和（`dstX1`，`dstY1`）限定的目标矩形。

像素具有半整数中心坐标。仅由 `BlitFramebuffer` 写入位于目标矩形内的像素。线性过滤采样（见下文）可能导致在读取源矩形之外的像素。

如果 `mask` 为零，则不复制任何缓冲区。

在传输颜色缓冲区时，值将从读取帧缓冲区的读取缓冲区取出，并写入绘制帧缓冲区的每个绘制缓冲区。从读取帧缓冲区取出的实际区域受限于正在传输的源缓冲区的交集，这可能包括由读取缓冲区选择的颜色缓冲区、深度缓冲区和/或模板缓冲区，具体取决于 `mask`。写入绘制帧缓冲区的实际区域受限于正在写入的目标缓冲区的交集，这可能包括多个绘制缓冲区、深度缓冲区和/或模板缓冲区，具体取决于 `mask`。无论源或目标区域是否由于这些限制而发生更改，传输的像素的缩放和偏移都会执行，就像没有这些限制一样。

如果源和目标矩形的尺寸不匹配，则源图像将被拉伸以适应目标矩形。`filter` 必须是 `LINEAR` 或 `NEAREST`，并指定在拉伸图像时要应用的插值方法。仅允许对颜色缓冲区使用 `LINEAR` 滤波。如果源和目标尺寸相同，则不应用任何滤波。如果源或目标矩形指定了负宽度或高度（X1 < X0 或 Y1 < Y0），则在相应方向上翻转图像。如果源和目标矩形为相同方向指定了负宽度或高度，则不执行翻转。如果选择了线性滤波，并且线性采样的规则要求在源缓冲区的边界之外进行采样，那么就好像执行了 `CLAMP_TO_EDGE` 纹理采样。如果选择了线性滤波，并且在指定的源区域的边界之外进行采样是必需的，但在源缓冲区的边界之内，实现可以选择在采样时夹紧或不夹紧。

如果源和目标缓冲区相同，并且源和目标矩形重叠，则复制操作的结果是未定义的，如第18.3节开头所述。

当从读取缓冲区获取值时，如果启用了 `FRAMEBUFFER_SRGB` 并且与读取缓冲区对应的帧缓冲区附件的 `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` 的值为 `SRGB`（见第9.2.3节），则根据等式8.17，将红色、绿色和蓝色分量从非线性sRGB颜色空间转换为线性sRGB颜色空间。

当值写入绘制缓冲区时，复制操作绕过大多数片段管线。影响复制的唯一片段操作是像素拥有权测试、剪刀测试和sRGB转换（见第17.3.7节）。颜色、深度和模板掩码（见第17.4.2节）将被忽略。

如果读取帧缓冲区是分层的（见第9.8节），则从第零层读取像素值。如果绘制帧缓冲区是分层的，则像素值将被写入第零层。

如果读取和绘制帧缓冲区都是分层的，那么复制操作仍然仅在第零层执行。

如果在 `mask` 中指定了一个缓冲区，而在读取和绘制帧缓冲区中都不存在，则相应的位将被默默地忽略。

如果读取和绘制缓冲区的颜色格式不匹配，并且 `mask` 包括 `COLOR_BUFFER_BIT`，则像素组将被转换以匹配目标格式。但是，只有当所有绘制颜色缓冲区都具有定点分量时，颜色才会被夹紧。如下所述，并不支持所有数据类型的格式转换。

如果读取帧缓冲区是多采样的（其有效值为 `SAMPLE_BUFFERS` 为一），而绘制帧缓冲区不是（其 `SAMPLE_BUFFERS` 的值为零），则在写入目标之前，将与源中每个像素位置对应的样本转换为单个样本。`filter` 被忽略。如果源格式是整数类型或模板值，则为每个像素选择单个样本的值。如果源格式是浮点或标准化类型，则每个像素的样本值以实现相关的方式解决。如果源格式是深度值，则样本值以实现相关的方式解决，其中结果将位于像素中的最小深度值和最大深度值之间。

如果读取帧缓冲区不是多采样的，而绘制帧缓冲区是多采样的，则源样本的值在每个目标样本中复制。

如果读取和绘制帧缓冲区都是多采样的，并且它们的有效值的 `SAMPLES` 是相同的，则样本将被复制到绘制帧缓冲区，而不进行修改（除非可能进行格式转换）。请注意，不能保证绘制缓冲区中的样本与读取缓冲区相同，因此使用这个新创建的缓冲区进行渲染可能会有几何裂缝或不正确的抗锯齿效果。这可能发生在帧缓冲区的大小不匹配或源和目标矩形未定义为相同（`X0`，`Y0`）和（`X1`，`Y1`）边界的情况下。

18.3.2 在图像之间复制

以下命令可用于在两个图像对象之间复制纹理数据的区域：

```
void CopyImageSubData(uint srcName, enum srcTarget,
                      int srcLevel, int srcX, int srcY, int srcZ,
                      uint dstName, enum dstTarget, int dstLevel, int dstX,
                      int dstY, int dstZ, sizei srcWidth, sizei srcHeight,
                      sizei srcDepth);
```

它用于在两个图像对象之间复制纹理数据的区域。图像对象可以是纹理或渲染缓冲区。

CopyImageSubData不执行通用的转换，比如缩放、调整大小、混合、颜色空间或格式转换。它应该被视为类似于CPU memcpy 的操作。CopyImageSubData可以在具有不同内部格式的图像之间复制，只要这些格式是兼容的。

CopyImageSubData还允许在某些类型的压缩和非压缩内部格式之间复制，如表18.4所述。此复制不执行即时压缩或解压缩。当从非压缩内部格式复制到压缩内部格式时，每个非压缩数据的纹素都变成一个压缩数据块。当从压缩内部格式复制到非压缩内部格式时，一个压缩数据块变成一个非压缩数据的纹素。非压缩格式的纹素大小必须与压缩格式的块大小相同。因此，可以在128位非压缩格式和使用8位4×4块的压缩格式之间或在64位非压缩格式和使用4位4×4块的压缩格式之间复制。

源对象由`srcName`和`srcTarget`标识。同样，目标对象由`dstName`和`dstTarget`标识。名称的解释取决于相应目标参数的值。如果目标参数是`RENDERBUFFER`，则该名称被解释为渲染缓冲区对象的名称。如果目标参数是纹理目标，则该名称被解释为纹理对象。接受所有非代理纹理目标，除了`TEXTURE_BUFFER`和表8.19中描述的立方体贴图面选择器之外。

`srcLevel`和`dstLevel`标识源和目标详细级别。对于纹理，这必须是纹理对象中的有效详细级别。对于渲染缓冲区，此值必须为零。

`srcX`、`srcY`和`srcZ`指定源纹理图像的一个`srcWidth x srcHeight x srcDepth`矩形子区域的左下纹素坐标。同样，`dstX`、`dstY`和`dstZ`指定目标纹理图像的子区域的坐标。源和目标子区域必须完全包含在相应图像对象的指定级别中。尽管对于压缩纹理格式，尺寸总是以纹素为单位指定的，但应注意，如果只有源和目标纹理中的一个是压缩的，那么在压缩图像中触及的纹素数量将是非压缩图像中的块大小的倍数。

一维数组的切片，二维数组，立方体贴图数组或三维纹理的切片，或立方体贴图纹理的面都是兼容的，前提是它们共享兼容的内部格式，多个切片或面可以通过一次调用进行复制，通过指定srcZ和dstZ指定起始切片，并通过srcDepth指定要复制的切片数。

立方体贴图纹理始终有六个面，根据表8.19中指定的顺序，这些面由零为基础的面索引进行选择。

对于`CopyImageSubData`，如果满足以下任何条件，则认为两个内部格式是兼容的：

- 格式相同，但如果格式相同而且是基本内部格式，则每个图像的实际内部格式（见第8.5节末尾）必须相同。
- 根据第8.18节定义的用于纹理视图的兼容性规则，格式被认为是兼容的。特别是，如果两个内部格式在表8.22的同一条目中列出，则它们被认为是兼容的。
- 一个格式是压缩的，另一个是非压缩的，并且表18.4中列出了这两种格式在同一行。

重叠复制会导致未定义的像素值，如第18.3节介绍中所述。

18.4 像素绘制和读取状态

用于像素操作的状态包括使用`PixelStore`设置的参数。此状态已在表8.1和18.1中总结。其他状态包括一个控制最终转换期间夹紧的三值整数。读取颜色夹紧的初始值为`FIXED_ONLY`。使用`PixelStore`设置的状态是GL客户端状态。

mesa-18 实现分析(core)

OpenGL 层

像素属性的存放及定义

mesa通过在gl_context内部定义一个gl_pixel_attrib 的Pixel实现

```
struct gl_context
{
   ...

   struct gl_pixel_attrib   Pixel;      /**< Pixel attributes */
   ...
}

/**
 * 像素属性组（GL_PIXEL_MODE_BIT）。
 */
struct gl_pixel_attrib
{
   GLenum16 ReadBuffer;    /**< 用于 glRead/CopyPixels() 的源缓冲区 */

   /*--- 开始像素传输状态 ---*/
   /* 字段按照它们应用的顺序排列... */

   /** 缩放和偏移（索引移位，偏移） */
   /*@{*/
   GLfloat RedBias, RedScale;
   GLfloat GreenBias, GreenScale;
   GLfloat BlueBias, BlueScale;
   GLfloat AlphaBias, AlphaScale;
   GLfloat DepthBias, DepthScale;
   GLint IndexShift, IndexOffset;
   /*@}*/

   /* 像素映射 */
   /* 注意：实际像素映射不是此属性组的一部分 */
   GLboolean MapColorFlag;
   GLboolean MapStencilFlag;

   /*--- 结束像素传输状态 ---*/

   /** glPixelZoom */
   GLfloat ZoomX, ZoomY;
};
```

颜色夹取 ClampColor

```
void GLAPIENTRY
_mesa_ClampColor(GLenum target, GLenum clamp)
{
   GET_CURRENT_CONTEXT(ctx);

   switch (target) {
   case GL_CLAMP_VERTEX_COLOR_ARB:
      if (ctx->API == API_OPENGL_CORE)
         goto invalid_enum;
        // opengl 兼容时下发光照状态
      FLUSH_VERTICES(ctx, _NEW_LIGHT);
      ctx->Light.ClampVertexColor = clamp;
      _mesa_update_clamp_vertex_color(ctx, ctx->DrawBuffer);
      break;
   case GL_CLAMP_FRAGMENT_COLOR_ARB:
      if (ctx->API == API_OPENGL_CORE)
         goto invalid_enum;
      FLUSH_VERTICES(ctx, _NEW_FRAG_CLAMP);
      ctx->Color.ClampFragmentColor = clamp;
      _mesa_update_clamp_fragment_color(ctx, ctx->DrawBuffer);
      break;
   case GL_CLAMP_READ_COLOR_ARB:
      ctx->Color.ClampReadColor = clamp;
      break;
   default:
      goto invalid_enum;
   }
   return;

invalid_enum:
   _mesa_error(ctx, GL_INVALID_ENUM, "glClampColor(%s)",
               _mesa_enum_to_string(target));
}
```

- 对于该接口，core规范值支持 GL_CLAMP_READ_COLOR_ARB，即CLAMP_READ_COLOR

选择缓冲区 ReadBuffer

```
void GLAPIENTRY
_mesa_ReadBuffer_no_error(GLenum buffer)
{
   GET_CURRENT_CONTEXT(ctx);
   read_buffer_no_error(ctx, ctx->ReadBuffer, buffer, "glReadBuffer");
        read_buffer(ctx, fb = ctx->ReadBuffer, buffer, caller, true);
            ...
            srcBuffer = read_buffer_enum_to_index(ctx, buffer) -> BUFFER_*;
```

```
            //更新当前colorreadbuffer
            _mesa_readbuffer(ctx, fb, buffer, srcBuffer);

            // 处理winsys framebuffer
            /* Call the device driver function only if fb is the bound read
buffer */
            if (fb == ctx->ReadBuffer) {
               if (ctx->Driver.ReadBuffer)
                  ctx->Driver.ReadBuffer(ctx, buffer);
                  [jump state_tracker  st_ReadBuffer]
            }
}

void
_mesa_readbuffer(struct gl_context *ctx, struct gl_framebuffer *fb,
                 GLenum buffer, gl_buffer_index bufferIndex)
{
   if ((fb == ctx->ReadBuffer) && _mesa_is_winsys_fbo(fb)) {
      /* Only update the per-context READ_BUFFER state if we're bound to
       * a window-system framebuffer.
       */
       // 这里将默认fb的像素读取缓冲更新
      ctx->Pixel.ReadBuffer = buffer;
   }

   fb->ColorReadBuffer = buffer;
   fb->_ColorReadBufferIndex = bufferIndex;

    // fb状态更新
   ctx->NewState |= _NEW_BUFFERS;
}
```

读取像素 ReadPixels

```
_mesa_ReadPixels_no_error(GLint x, GLint y, GLsizei width, GLsizei height,
                          GLenum format, GLenum type, GLvoid *pixels)
{
   _mesa_ReadnPixelsARB_no_error(x, y, width, height, format, type, INT_MAX,
                                 pixels);
        read_pixels(x, y, width, height, format, type, bufSize, pixels, true);
}


static ALWAYS_INLINE void
read_pixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format,
            GLenum type, GLsizei bufSize, GLvoid *pixels, bool no_error)
{
   struct gl_renderbuffer *rb;
   struct gl_pixelstore_attrib clippedPacking;

   if (ctx->NewState)
      _mesa_update_state(ctx);

    // 获取当前readbuffer的附件的Renderbuffer,里面teximage保存了图像数据
```

```
   rb = _mesa_get_read_renderbuffer_for_format(ctx, format);
       const struct gl_framebuffer *rfb = ctx->ReadBuffer;

       if (_mesa_is_color_format(format)) {
          return rfb->Attachment[rfb->_ColorReadBufferIndex].Renderbuffer;
       } else if (_mesa_is_depth_format(format) ||
                  _mesa_is_depthstencil_format(format)) {
          return rfb->Attachment[BUFFER_DEPTH].Renderbuffer;
       } else {
          return rfb->Attachment[BUFFER_STENCIL].Renderbuffer;
       }



   /* Do all needed clipping here, so that we can forget about it later */
   clippedPacking = ctx->Pack;
   if (!_mesa_clip_readpixels(ctx, &x, &y, &width, &height, &clippedPacking))
      return; /* nothing to do */

   ctx->Driver.ReadPixels(ctx, x, y, width, height,
                          format, type, &clippedPacking, pixels);
        [jump state_tracker st_ReadPixels]
}
```

- 当维度不匹配时进行维度裁剪

裁剪像素维度

```
/**
 * 对于glReadPixels执行裁剪。调整图像的窗口位置和大小，以及pack skipPixels、skipRows和
rowLength，
 * 使图像区域完全位于窗口边界内。注意：这与_mesalip_drawpixels()不同，因为忽略了剪刀框，
 * 我们使用当前readbuffer表面或附加图像的边界。
 *
 * \return  GL_TRUE：如果要读取的区域在范围内
 *          GL_FALSE：如果区域完全超出范围（无需读取）
 */
GLboolean
_mesa_clip_readpixels(const struct gl_context *ctx,
                      GLint *srcX, GLint *srcY,
                      GLsizei *width, GLsizei *height,
                      struct gl_pixelstore_attrib *pack)
{
   const struct gl_framebuffer *buffer = ctx->ReadBuffer;
   struct gl_renderbuffer *rb = buffer->_ColorReadBuffer;
   GLsizei clip_width;
   GLsizei clip_height;

   if (rb) {
      clip_width = rb->Width;
      clip_height = rb->Height;
   } else {
      clip_width = buffer->Width;
      clip_height = buffer->Height;
```

```
   }

   if (pack->RowLength == 0) {
      pack->RowLength = *width;
   }

   /* 左侧裁剪 */
   if (*srcX < 0) {
      pack->SkipPixels += (0 - *srcX);
      *width -= (0 - *srcX);
      *srcX = 0;
   }
   /* 右侧裁剪 */
   if (*srcX + *width > clip_width)
      *width -= (*srcX + *width - clip_width);

   if (*width <= 0)
      return GL_FALSE;

   /* 底部裁剪 */
   if (*srcY < 0) {
      pack->SkipRows += (0 - *srcY);
      *height -= (0 - *srcY);
      *srcY = 0;
   }
   /* 顶部裁剪 */
   if (*srcY + *height > clip_height)
      *height -= (*srcY + *height - clip_height);

   if (*height <= 0)
      return GL_FALSE;

   return GL_TRUE;
}
```

复制像素

复制像素矩形 BlitFramebuffer

```


/**
 * Blit rectangular region, optionally from one framebuffer to another.
 *
 * Note, if the src buffer is multisampled and the dest is not, this is
 * when the samples must be resolved to a single color.
 */
void GLAPIENTRY
_mesa_BlitFramebuffer_no_error(GLint srcX0, GLint srcY0, GLint srcX1,
                               GLint srcY1, GLint dstX0, GLint dstY0,
                               GLint dstX1, GLint dstY1,
                               GLbitfield mask, GLenum filter)
{
   GET_CURRENT_CONTEXT(ctx);
```

```
   blit_framebuffer(ctx, ctx->ReadBuffer, ctx->DrawBuffer,
                    srcX0, srcY0, srcX1, srcY1,
                    dstX0, dstY0, dstX1, dstY1,
                    mask, filter, true, "glBlitFramebuffer");
}


// 在OpenGL中进行帧缓冲拷贝的函数
static ALWAYS_INLINE void
blit_framebuffer(struct gl_context *ctx,
                 struct gl_framebuffer *readFb, struct gl_framebuffer *drawFb,
                 GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1,
                 GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1,
                 GLbitfield mask, GLenum filter, bool no_error, const char *func)
{
   // 刷新顶点数据缓冲区
   FLUSH_VERTICES(ctx, 0);

   // 如果readFb或drawFb为空，则通常不会发生，但可能在未来支持没有可绘制区域的MakeCurrent()
时会用到。
   if (!readFb || !drawFb) {
      return;
   }

   // 更新readFb和drawFb的完整性状态。
   _mesa_update_framebuffer(ctx, readFb, drawFb);

   // 确保drawFb有一个已初始化的边界框。
   _mesa_update_draw_buffer_bounds(ctx, drawFb);

   ...

   // 获取颜色读/写渲染缓冲区
   if (mask & GL_COLOR_BUFFER_BIT) {
      const GLuint numColorDrawBuffers = drawFb->_NumColorDrawBuffers;
      const struct gl_renderbuffer *colorReadRb = readFb->_ColorReadBuffer;

      // 根据EXT_framebuffer_object规范：
      //     "如果在<mask>中指定了一个缓冲区，但在读取和绘制帧缓冲区中都不存在，
      //     则相应的位将被静默忽略。"
      if (!colorReadRb || numColorDrawBuffers == 0) {
         mask &= ~GL_COLOR_BUFFER_BIT;
      } else if (!no_error) {
         // 验证颜色缓冲区
         if (!validate_color_buffer(ctx, readFb, drawFb, filter, func))
            return;
      }
   }

   if (mask & GL_STENCIL_BUFFER_BIT) {
      struct gl_renderbuffer *readRb =
         readFb->Attachment[BUFFER_STENCIL].Renderbuffer;
      struct gl_renderbuffer *drawRb =
         drawFb->Attachment[BUFFER_STENCIL].Renderbuffer;
```

```
      // 根据EXT_framebuffer_object规范：
      //     "如果在<mask>中指定了一个缓冲区，但在读取和绘制帧缓冲区中都不存在，
      //     则相应的位将被静默忽略。"
      if ((readRb == NULL) || (drawRb == NULL)) {
         mask &= ~GL_STENCIL_BUFFER_BIT;
      } else if (!no_error) {
         // 验证模板缓冲区
         if (!validate_stencil_buffer(ctx, readFb, drawFb, func))
            return;
      }
   }

   if (mask & GL_DEPTH_BUFFER_BIT) {
      struct gl_renderbuffer *readRb =
         readFb->Attachment[BUFFER_DEPTH].Renderbuffer;
      struct gl_renderbuffer *drawRb =
         drawFB->Attachment[BUFFER_DEPTH].Renderbuffer;

      // 根据EXT_framebuffer_object规范：
      //     "如果在<mask>中指定了一个缓冲区，但在读取和绘制帧缓冲区中都不存在，
      //     则相应的位将被静默忽略。"
      if ((readRb == NULL) || (drawRb == NULL)) {
         mask &= ~GL_DEPTH_BUFFER_BIT;
      } else if (!no_error) {
         // 验证深度缓冲区
         if (!validate_depth_buffer(ctx, readFb, drawFb, func))
            return;
      }
   }

   // 如果没有掩码，或者源和目标矩形的宽度或高度为零，则返回
   if (!mask ||
       (srcX1 - srcX0) == 0 || (srcY1 - srcY0) == 0 ||
       (dstX1 - dstX0) == 0 || (dstY1 - dstY0) == 0) {
      return;
   }

   // 确保驱动程序支持BlitFramebuffer
   ctx->Driver.BlitFramebuffer(ctx, readFb, drawFb,
                               srcX0, srcY0, srcX1, srcY1,
                               dstX0, dstY0, dstX1, dstY1,
                               mask, filter);
            [jump stack_tracker st_BlitFramebuffer]
```

在图像之间复制 CopyImageSubData

```
void GLAPIENTRY
_mesa_CopyImageSubData_no_error(GLuint srcName, GLenum srcTarget, GLint srcLevel,
                                GLint srcX, GLint srcY, GLint srcZ,
                                GLuint dstName, GLenum dstTarget, GLint dstLevel,
                                GLint dstX, GLint dstY, GLint dstZ,
                                GLsizei srcWidth, GLsizei srcHeight, GLsizei
srcDepth)
```

```
{
   struct gl_texture_image *srcTexImage, *dstTexImage;
   struct gl_renderbuffer *srcRenderbuffer, *dstRenderbuffer;

   GET_CURRENT_CONTEXT(ctx);

    // 获取srcTexImage 或者srcRenderbuffer
   prepare_target(ctx, srcName, srcTarget, srcLevel, srcZ, &srcTexImage,
                  &srcRenderbuffer);
        ..................................
           if (target == GL_RENDERBUFFER) {
              struct gl_renderbuffer *rb = _mesa_lookup_renderbuffer(ctx, name);

              *renderbuffer = rb;
              *texImage = NULL;
           } else {
              struct gl_texture_object *texObj = _mesa_lookup_texture(ctx, name);

              if (target == GL_TEXTURE_CUBE_MAP) {
                 *texImage = texObj->Image[z][level];
              }
              else {
                 *texImage = _mesa_select_tex_image(texObj, target, level);
              }

              *renderbuffer = NULL;
           }

   prepare_target(ctx, dstName, dstTarget, dstLevel, dstZ, &dstTexImage,
                  &dstRenderbuffer);

   copy_image_subdata(ctx, srcTexImage, srcRenderbuffer, srcX, srcY, srcZ,
                      srcLevel, dstTexImage, dstRenderbuffer, dstX, dstY, dstZ,
                      dstLevel, srcWidth, srcHeight, srcDepth);
}

static void
copy_image_subdata(struct gl_context *ctx,
                   struct gl_texture_image *srcTexImage,
                   struct gl_renderbuffer *srcRenderbuffer,
                   int srcX, int srcY, int srcZ, int srcLevel,
                   struct gl_texture_image *dstTexImage,
                   struct gl_renderbuffer *dstRenderbuffer,
                   int dstX, int dstY, int dstZ, int dstLevel,
                   int srcWidth, int srcHeight, int srcDepth)
{
   /* loop over 2D slices/faces/layers */
   for (int i = 0; i < srcDepth; ++i) {
      int newSrcZ = srcZ + i;
      int newDstZ = dstZ + i;

      if (srcTexImage &&
          srcTexImage->TexObject->Target == GL_TEXTURE_CUBE_MAP) {
         /* need to update srcTexImage pointer for the cube face */
         assert(srcZ + i < MAX_FACES);
         srcTexImage = srcTexImage->TexObject->Image[srcZ + i][srcLevel];
```

```
         assert(srcTexImage);
         newSrcZ = 0;
      }

      if (dstTexImage &&
          dstTexImage->TexObject->Target == GL_TEXTURE_CUBE_MAP) {
         /* need to update dstTexImage pointer for the cube face */
         assert(dstZ + i < MAX_FACES);
         dstTexImage = dstTexImage->TexObject->Image[dstZ + i][dstLevel];
         assert(dstTexImage);
         newDstZ = 0;
      }

      ctx->Driver.CopyImageSubData(ctx,
                                   srcTexImage, srcRenderbuffer,
                                   srcX, srcY, newSrcZ,
                                   dstTexImage, dstRenderbuffer,
                                   dstX, dstY, newDstZ,
                                   srcWidth, srcHeight);
            [jump state_tracker st_CopyImageSubData]
   }
}
```

state tracker

BlitFramebuffer/ st_BlitFramebuffer

接口

```
// st_cb_blit.c

void
st_init_blit_functions(struct dd_function_table *functions)
{
   functions->BlitFramebuffer = st_BlitFramebuffer;
}
```

st_BlitFramebuffer

```
static void
st_BlitFramebuffer(struct gl_context *ctx,
                   struct gl_framebuffer *readFB,
                   struct gl_framebuffer *drawFB,
                   GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1,
                   GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1,
                   GLbitfield mask, GLenum filter)
{
   const GLbitfield depthStencil = (GL_DEPTH_BUFFER_BIT |
                                    GL_STENCIL_BUFFER_BIT);
   struct st_context *st = st_context(ctx);
   const uint pFilter = ((filter == GL_NEAREST)
```

```
                      ? PIPE_TEX_FILTER_NEAREST
                      : PIPE_TEX_FILTER_LINEAR);
struct {
   GLint srcX0, srcY0, srcX1, srcY1;
   GLint dstX0, dstY0, dstX1, dstY1;
} clip;
struct pipe_blit_info blit;

st_manager_validate_framebuffers(st);

/* Make sure bitmap rendering has landed in the framebuffers */
st_flush_bitmap_cache(st);
st_invalidate_readpix_cache(st);

clip.xxx= xxx;

 // blit 维度裁剪
if (!_mesa_clip_blit(ctx, readFB, drawFB,
                     &clip.srcX0, &clip.srcY0, &clip.srcX1, &clip.srcY1,
                     &clip.dstX0, &clip.dstY0, &clip.dstX1, &clip.dstY1)) {
   return; /* nothing to draw/blit */
}
memset(&blit, 0, sizeof(struct pipe_blit_info));
blit.scissor_enable =...;

if (st_fb_orientation(drawFB) == Y_0_TOP) {
     ...
}
if (blit.scissor_enable) {
   blit.scissor.maxx = ...;
   ...
}

if (st_fb_orientation(readFB) == Y_0_TOP) {
     ...
}

if (srcY0 > srcY1 && dstY0 > dstY1) {
     ...;
}

blit.src.box.depth = 1;
blit.dst.box.depth = 1;

/* Destination dimensions have to be positive: */
if (dstX0 < dstX1) {
   blit.dst.box.x = dstX0;
   blit.src.box.x = srcX0;
   blit.dst.box.width = dstX1 - dstX0;
   blit.src.box.width = srcX1 - srcX0;
else {
     ...
}
```

```
if (drawFB != ctx->WinSysDrawBuffer)
   st_window_rectangles_to_blit(ctx, &blit);

blit.filter = pFilter;
blit.render_condition_enable = TRUE;
blit.alpha_blend = FALSE;

if (mask & GL_COLOR_BUFFER_BIT) {
   struct gl_renderbuffer_attachment *srcAtt =
      &readFB->Attachment[readFB->_ColorReadBufferIndex];
   GLuint i;


   // 填充blit_info

   for (i = 0; i < drawFB->_NumColorDrawBuffers; i++) {
      struct st_renderbuffer *dstRb =
         st_renderbuffer(drawFB->_ColorDrawBuffers[i]);

      if (dstRb) {
         struct pipe_surface *dstSurf;

         st_update_renderbuffer_surface(st, dstRb);

         dstSurf = dstRb->surface;

         if (dstSurf) {
            ...

            st->pipe->blit(st->pipe, &blit);
            dstRb->defined = true; /* front buffer tracking */
         }
      }
   }
}

if (mask & depthStencil) {
   /* depth and/or stencil blit */

   /* get src/dst depth surfaces */
   struct st_renderbuffer *srcDepthRb =
      st_renderbuffer(readFB->Attachment[BUFFER_DEPTH].Renderbuffer);
   struct st_renderbuffer *dstDepthRb = ...;

   ...


   if (_mesa_has_depthstencil_combined(readFB) &&
       _mesa_has_depthstencil_combined(drawFB)) {
       ...

      st->pipe->blit(st->pipe, &blit);
   }
   else {
      /* blitting depth and stencil separately */
```

```
            if (mask & GL_DEPTH_BUFFER_BIT) {
               //处理深度附件

               // 填充blit_info
               ...

               st->pipe->blit(st->pipe, &blit);
            }

            if (mask & GL_STENCIL_BUFFER_BIT) {
               // 处理模板附件

               st->pipe->blit(st->pipe, &blit);
            }
         }
      }
   }
```

- 关于帧缓冲方向

```
// 获取帧缓冲的方向
static inline GLuint
st_fb_orientation(const struct gl_framebuffer *fb)
{
   // 如果帧缓冲存在且是窗口系统帧缓冲对象
   if (fb && _mesa_is_winsys_fbo(fb)) {
      /* 绘制到窗口（屏幕缓冲区）。
       *
       * 反转Y轴以垂直翻转图像。
       * 在进行视口变换之前，NDC坐标的Y坐标在范围[y=-1=bottom, y=1=top]内。
       * 硬件窗口坐标在范围[y=0=top, y=H-1=bottom]内，其中H是窗口的高度。
       * 使用视口变换来反转Y。
       */
      return Y_0_TOP;
   }
   // 否则，绘制到用户创建的FBO（很可能是一个纹理）。
   else {
      /* 对于纹理，T=0=底部，因此根据推论，Y=0=底部用于渲染。*/
      return Y_0_BOTTOM;
   }
}
```

fbo 中用于像素读取的接口

由于这些接口在readpixels中比较常用， 放在这里

```
void
st_init_fbo_functions(struct dd_function_table *functions)
{
   ...
   functions->ReadBuffer = st_ReadBuffer;

   functions->MapRenderbuffer = st_MapRenderbuffer;
   functions->UnmapRenderbuffer = st_UnmapRenderbuffer;
}
```

fbo 选择读取缓冲 ReadBuffer/st_ReadBuffer

ReadBuffer接口同fbo分析

```
/**
 * Called via glReadBuffer.  As with st_DrawBufferAllocate, we use this
 * function to check if we need to allocate a renderbuffer on demand.
 */
static void
st_ReadBuffer(struct gl_context *ctx, GLenum buffer)
{
   struct st_context *st = st_context(ctx);
   struct gl_framebuffer *fb = ctx->ReadBuffer;

   (void) buffer;

   /* Check if we need to allocate a front color buffer.
    * Front buffers are often allocated on demand (other color buffers are
    * always allocated in advance).
    */
   if ((fb->_ColorReadBufferIndex == BUFFER_FRONT_LEFT ||
        fb->_ColorReadBufferIndex == BUFFER_FRONT_RIGHT) &&
       fb->Attachment[fb->_ColorReadBufferIndex].Type == GL_NONE) {
      assert(_mesa_is_winsys_fbo(fb));
      /* add the buffer */
      st_manager_add_color_renderbuffer(st, fb, fb->_ColorReadBufferIndex);
      _mesa_update_state(ctx);
      st_validate_state(st, ST_PIPELINE_UPDATE_FRAMEBUFFER);
   }
}
```

- 这个接口只用在默认fb

fbo MapRenderbuffer /st_MapRenderbuffer

```
/**
 * Called via ctx->Driver.MapRenderbuffer.
 */
static void
st_MapRenderbuffer(struct gl_context *ctx,
                   struct gl_renderbuffer *rb,
                   GLuint x, GLuint y, GLuint w, GLuint h,
```

```
                    GLbitfield mode,
                    GLubyte **mapOut, GLint *rowStrideOut,
                    bool flip_y)
{
   struct st_context *st = st_context(ctx);
   struct st_renderbuffer *strb = st_renderbuffer(rb);
   struct pipe_context *pipe = st->pipe;
   const GLboolean invert = rb->Name == 0;
   GLuint y2;
   GLubyte *map;

   /* driver does not support GL_FRAMEBUFFER_FLIP_Y_MESA */
   assert((rb->Name == 0) == flip_y);

    //对于winsys是软件分配的 see st_new_renderbuffer_fb
   if (strb->software) {
      /* software-allocated renderbuffer (probably an accum buffer) */
      if (strb->data) {
         GLint bpp = _mesa_get_format_bytes(strb->Base.Format);
         GLint stride = _mesa_format_row_stride(strb->Base.Format,
                                                strb->Base.Width);
         *mapOut = (GLubyte *) strb->data + y * stride + x * bpp;
         *rowStrideOut = stride;
      }
      else {
         *mapOut = NULL;
         *rowStrideOut = 0;
      }
      return;
   }

   /* Check for unexpected flags */
   assert((mode & ~(GL_MAP_READ_BIT |
                    GL_MAP_WRITE_BIT |
                    GL_MAP_INVALIDATE_RANGE_BIT)) == 0);

   const enum pipe_transfer_usage transfer_flags =
      st_access_flags_to_transfer_flags(mode, false);

   /* Note: y=0=bottom of buffer while y2=0=top of buffer.
    * 'invert' will be true for window-system buffers and false for
    * user-allocated renderbuffers and textures.
    */
   if (invert)
      y2 = strb->Base.Height - y - h;
   else
      y2 = y;

    map = pipe_transfer_map(pipe,
                            strb->texture,
                            strb->surface->u.tex.level,
                            strb->surface->u.tex.first_layer,
                            transfer_flags, x, y2, w, h, &strb->transfer);
   if (map) {
      if (invert) {
         *rowStrideOut = -(int) strb->transfer->stride;
```

```
         map += (h - 1) * strb->transfer->stride;
      }
      else {
         *rowStrideOut = strb->transfer->stride;
      }
      *mapOut = map;
   }
   else {
      *mapOut = NULL;
      *rowStrideOut = 0;
   }
}
```

- 当对于默认fb时， 直接进行计算处理， 否则该接口同tranfer_map分析一文类似，而返回参数通过mapOut返回

fbo UnmapRenderbuffer /st_UnmapRenderbuffer

```
/**
 * Called via ctx->Driver.UnmapRenderbuffer.
 */
static void
st_UnmapRenderbuffer(struct gl_context *ctx,
                     struct gl_renderbuffer *rb)
{
   struct st_context *st = st_context(ctx);
   struct st_renderbuffer *strb = st_renderbuffer(rb);
   struct pipe_context *pipe = st->pipe;

   if (strb->software) {
      /* software-allocated renderbuffer (probably an accum buffer) */
      return;
   }

   pipe_transfer_unmap(pipe, strb->transfer);
   strb->transfer = NULL;
}
```

- 用法与MapRenderbuffer一致,处理了默认fb情况

像素读取接口

```
void st_init_readpixels_functions(struct dd_function_table *functions)
{
   functions->ReadPixels = st_ReadPixels;
}
```

读取像素 ReadPixels /st_ReadPixels

st_ReadPixels的实现机理恰好是teximage 的打包形式，逻辑结构相似， see texture_format.pdf

```
/**
 * 使用位块传输（blit）将读取缓冲区复制到与指定格式和类型相匹配的纹理格式，
 * 然后使用memcpy进行快速读取。我们可以在这里执行任意的X/Y/Z/W/0/1混合，
 * 只要有与混合匹配的格式即可。
 *
 * 如果没有这样的格式可用，就回退到_mesa_readpixels。
 *
 * 注意：一些驱动程序在纹理上传/下载期间使用位块传输在瓦片和线性纹理布局之间进行转换，
 * 所以我们在这里执行的位块传输在这种情况下应该是免费的。
 */
static void
st_ReadPixels(struct gl_context *ctx, GLint x, GLint y,
              GLsizei width, GLsizei height,
              GLenum format, GLenum type,
              const struct gl_pixelstore_attrib *pack = clippedPacking,
              void *pixels)
{
   struct st_context *st = st_context(ctx);
   struct gl_renderbuffer *rb =
         _mesa_get_read_renderbuffer_for_format(ctx, format);
   struct st_renderbuffer *strb = st_renderbuffer(rb);
   struct pipe_context *pipe = st->pipe;
   struct pipe_screen *screen = pipe->screen;
   struct pipe_resource *src;
   struct pipe_resource *dst = NULL;
   enum pipe_format dst_format, src_format;
   unsigned bind;
   struct pipe_transfer *tex_xfer;
   ubyte *map = NULL;
   int dst_x, dst_y;

   /* 验证状态（确保我们有最新的帧缓冲表面），
    * 并在读取之前刷新位图缓存。*/
   st_validate_state(st, ST_PIPELINE_UPDATE_FRAMEBUFFER);
   st_flush_bitmap_cache(st);

    // context创建时通过get_param向驱动获取
PIPE_CAP_PREFER_BLIT_BASED_TEXTURE_TRANSFER参数的支持，radeonsi为真
   if (!st->prefer_blit_based_texture_transfer) {
      goto fallback;
   }

   /* 这必须在状态验证之后完成。 */
   src = strb->texture;

   /* XXX 某些驱动程序由于某些驱动程序中不完整的模板位块传输实现而回退到深度模板格式。 */
   if (format == GL_DEPTH_STENCIL) {
      goto fallback;
   }

   /* 如果基本内部格式与纹理格式不匹配，则必须使用慢路径。 */
```

```
   if (rb->_BaseFormat !=
       _mesa_get_format_base_format(rb->Format)) {
      goto fallback;
   }

   if (_mesa_readpixels_needs_slow_path(ctx, format, type, GL_TRUE)) {
      goto fallback;
   }

   /* 将源格式转换为ReadPixels期望的格式，并查看是否支持。 */
   src_format = util_format_linear(src->format);
   src_format = util_format_luminance_to_red(src_format);
   src_format = util_format_intensity_to_red(src_format);

   if (!src_format ||
       !screen->is_format_supported(screen, src_format, src->target,
                                    src->nr_samples, src->nr_storage_samples,
                                    PIPE_BIND_SAMPLER_VIEW)) {
      goto fallback;
   }

   if (format == GL_DEPTH_COMPONENT || format == GL_DEPTH_STENCIL)
      bind = PIPE_BIND_DEPTH_STENCIL;
   else
      bind = PIPE_BIND_RENDER_TARGET;

   /* 选择与格式+类型组合最匹配的目标格式。 */
   dst_format = st_choose_matching_format(st, bind, format, type,
                                          pack->SwapBytes);
   if (dst_format == PIPE_FORMAT_NONE) {
      goto fallback;
   }

    // 如果存在pbo，
   if (st->pbo.download_enabled && _mesa_is_bufferobj(pack->BufferObj)) {
      if (try_pbo_readpixels(st, strb,
                             st_fb_orientation(ctx->ReadBuffer) == Y_0_TOP,
                             x, y, width, height,
                             src_format, dst_format,
                             pack, pixels))
         return;
   }

   if (needs_integer_signed_unsigned_conversion(ctx, format, type)) {
      goto fallback;
   }

   /* 缓存用于连续的ReadPixels的临时纹理，以避免CPU-GPU同步开销。 */
   dst = try_cached_readpixels(st, strb,
                               st_fb_orientation(ctx->ReadBuffer) == Y_0_TOP,
                               width, height, format, src_format, dst_format);
      if (ST_DEBUG & DEBUG_NOREADPIXCACHE)
         return NULL;

      /* Decide whether to trigger the cache. */
      if (!st->readpix_cache.cache) {
```

```
            // ALWAYS_READPIXELS_CACHE 为宏定义默认是false
           if (!strb->use_readpix_cache && !ALWAYS_READPIXELS_CACHE) {
           /* Fill the cache */
           st->readpix_cache.cache = blit_to_staging(st, strb, invert_y,
                                                     0, 0,
                                                     strb->Base.Width,
                                                     strb->Base.Height, format,
                                                     src_format, dst_format);
        }
        /* Return an owning reference to stay consistent with the non-cached path
*/
        pipe_resource_reference(&dst, st->readpix_cache.cache);

        return dst;


   if (dst) {
      dst_x = x;
      dst_y = y;
   } else {
      /* 查看纹理格式是否已经与格式和类型匹配，
       * 如果是，则可能使用基于memcpy的快速路径，我们不必进行位块传输。 */
      if (_mesa_format_matches_format_and_type(rb->Format, format,
                                               type, pack->SwapBytes, NULL)) {
         goto fallback;
      }

      dst = blit_to_staging(st, strb,
                            st_fb_orientation(ctx->ReadBuffer) == Y_0_TOP,
                            x, y, width, height, format,
                            src_format, dst_format);
      if (!dst)
         goto fallback;

      dst_x = 0;
      dst_y = 0;
   }

   /* 映射资源 */
   pixels = _mesa_map_pbo_dest(ctx, pack, pixels);

   map = pipe_transfer_map_3d(pipe, dst, 0, PIPE_TRANSFER_READ,
                              dst_x, dst_y, 0, width, height, 1, &tex_xfer);
   if (!map) {
      _mesa_unmap_pbo_dest(ctx, pack);
      pipe_resource_reference(&dst, NULL);
      goto fallback;
   }

   /* 将数据memcpy到用户缓冲区 */
   {
      const uint bytesPerRow = width * util_format_get_blocksize(dst_format);
      const int destStride = _mesa_image_row_stride(pack, width, format, type);
      char *dest = _mesa_image_address2d(pack, pixels,
                                         width, height, format,
                                         type, 0, 0);
```

```
      if (tex_xfer->stride == bytesPerRow && destStride == bytesPerRow) {
         memcpy(dest, map, bytesPerRow * height);
      } else {
         GLuint row;

         for (row = 0; row < (unsigned) height; row++) {
            memcpy(dest, map, bytesPerRow);
            map += tex_xfer->stride;
            dest += destStride;
         }
      }
   }

   pipe_transfer_unmap(pipe, tex_xfer);
   _mesa_unmap_pbo_dest(ctx, pack);
   pipe_resource_reference(&dst, NULL);
   return;

fallback:
   _mesa_readpixels(ctx, x, y, width, height, format, type, pack, pixels);
}
```

- 该接口通过两部分实现，一是使用blit快速传输，二是使用_mesa_readpixels , 这个与teximage的机理有雷同之处

通过blit读取

在条件满足时，使用blit会根据是否使用pbo通过try_pbo_readpixels 读取像素; 如果不是pbo， 会根据是否禁用pixelcache 需要调用try_cached_readpixels 进行像素缓存读取，当然缓存不存在时会首先通过blit_to_staging创建一个暂存纹理当作缓存,然后，返回作为引用;当不使用缓存的情况下，其实也是使用blit_to_staging 创建一个暂存纹理实现

关于 blit_to_staging,

```
/**
 * Create a staging texture and blit the requested region to it.
 */
static struct pipe_resource *
blit_to_staging(struct st_context *st, struct st_renderbuffer *strb,
                  bool invert_y,
                  GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format,
                  enum pipe_format src_format, enum pipe_format dst_format)
{
   struct pipe_blit_info blit;
   ...

   dst = screen->resource_create(screen, &dst_templ);
   if (!dst)
      return NULL;

   memset(&blit, 0, sizeof(blit));
```

```
   blit.src.resource = strb->texture;
   blit.src.xxx = ...;

   /* blit */
   st->pipe->blit(st->pipe, &blit);
   return dst;
}
```

- see si_blit

通过_mesa_readpixels读取像素

```
/**
 * Software fallback routine for ctx->Driver.ReadPixels().
 * By time we get here, all error checking will have been done.
 */
void
_mesa_readpixels(struct gl_context *ctx,
                 GLint x, GLint y, GLsizei width, GLsizei height,
                 GLenum format, GLenum type,
                 const struct gl_pixelstore_attrib *packing,
                 GLvoid *pixels)
{
   if (ctx->NewState)
      _mesa_update_state(ctx);

    // for pbo
   pixels = _mesa_map_pbo_dest(ctx, packing, pixels);

   if (pixels) {
      /* Try memcpy first. */
      if (readpixels_memcpy(ctx, x, y, width, height, format, type,
                            pixels, packing)) {
         _mesa_unmap_pbo_dest(ctx, packing);
         return;
      }

      /* Otherwise take the slow path. */
      switch (format) {
      case GL_STENCIL_INDEX:
         read_stencil_pixels(ctx, x, y, width, height, type, pixels,
                             packing);
         break;
      case GL_DEPTH_COMPONENT:
         read_depth_pixels(ctx, x, y, width, height, type, pixels,
                           packing);
         break;
      case GL_DEPTH_STENCIL_EXT:
         read_depth_stencil_pixels(ctx, x, y, width, height, type, pixels,
                                   packing);
         break;
      default:
         /* all other formats should be color formats */
         read_rgba_pixels(ctx, x, y, width, height, format, type, pixels,
```

```
                           packing);
      }

      _mesa_unmap_pbo_dest(ctx, packing);
   }
}
```

- 该函数首先尝试通过readpixels_memcpy拷贝形式读取像素，否则针对不同的缓冲附件处理

readpixels_memcpy

```
static GLboolean
readpixels_memcpy(struct gl_context *ctx,
                  GLint x, GLint y,
                  GLsizei width, GLsizei height,
                  GLenum format, GLenum type,
                  GLvoid *pixels,
                  const struct gl_pixelstore_attrib *packing)
{
   struct gl_renderbuffer *rb =
         _mesa_get_read_renderbuffer_for_format(ctx, format);
         ------------------------------------------------------
              const struct gl_framebuffer *rfb = ctx->ReadBuffer;
              if (_mesa_is_color_format(format)) {
                 return rfb->Attachment[rfb->_ColorReadBufferIndex].Renderbuffer;
              } else if (_mesa_is_depth_format(format) ||
                         _mesa_is_depthstencil_format(format)) {
                 return rfb->Attachment[BUFFER_DEPTH].Renderbuffer;
              } else {
                 return rfb->Attachment[BUFFER_STENCIL].Renderbuffer;
              }

   GLubyte *dst, *map;
   int dstStride, stride, j, texelBytes, bytesPerRow;

   /* Fail if memcpy cannot be used. */
   if (!readpixels_can_use_memcpy(ctx, format, type, packing)) {
      return GL_FALSE;
   }

   dstStride = _mesa_image_row_stride(packing, width, format, type);
   dst = (GLubyte *) _mesa_image_address2d(packing, pixels, width, height,
                       format, type, 0, 0);

    // 获取map地址
   ctx->Driver.MapRenderbuffer(ctx, rb, x, y, width, height, GL_MAP_READ_BIT,
                   &map, &stride, ctx->ReadBuffer->FlipY);
                   [jump state_tracker st_MapRenderbuffer]
   if (!map) {
      _mesa_error(ctx, GL_OUT_OF_MEMORY, "glReadPixels");
      return GL_TRUE;  /* don't bother trying the slow path */
   }

   texelBytes = _mesa_get_format_bytes(rb->Format);
```

```
   bytesPerRow = texelBytes * width;

   /* memcpy*/
   if (dstStride == stride && dstStride == bytesPerRow) {
      memcpy(dst, map, bytesPerRow * height);
   } else {
      for (j = 0; j < height; j++) {
         memcpy(dst, map, bytesPerRow);
         dst += dstStride;
         map += stride;
      }
   }

   ctx->Driver.UnmapRenderbuffer(ctx, rb);
   return GL_TRUE;
}
```

- 该接口只有mesa_formt 和 formattype类型相一致时才可直接拷贝处理:

针对不同附件 读取像素

由于格式的不匹配所以这时需要针对不同格式附件进行格式转换

stencil

```
/**
 * Read pixels for format=GL_STENCIL_INDEX.
 */
static void
read_stencil_pixels( struct gl_context *ctx,
                     GLint x, GLint y,
                     GLsizei width, GLsizei height,
                     GLenum type, GLvoid *pixels,
                     const struct gl_pixelstore_attrib *packing )
{
   struct gl_framebuffer *fb = ctx->ReadBuffer;
   struct gl_renderbuffer *rb = fb->Attachment[BUFFER_STENCIL].Renderbuffer;
   GLint j;
   GLubyte *map, *stencil;
   GLint stride;

   if (!rb)
      return;

   ctx->Driver.MapRenderbuffer(ctx, rb, x, y, width, height, GL_MAP_READ_BIT,
                  &map, &stride, fb->FlipY);
   if (!map) {
      _mesa_error(ctx, GL_OUT_OF_MEMORY, "glReadPixels");
      return;
   }

   stencil = malloc(width * sizeof(GLubyte));

   if (stencil) {
      /* process image row by row */
```

```
      for (j = 0; j < height; j++) {
         GLvoid *dest;

         _mesa_unpack_ubyte_stencil_row(rb->Format, width, map, stencil);

         dest = _mesa_image_address2d(packing, pixels, width, height,

         _mesa_pack_stencil_span(ctx, width, type, dest, stencil, packing);

            GLubyte *stencil = malloc(n * sizeof(GLubyte));

            // IndexShift 由glPixelTransfer标志确定，在兼容模式下使用：
            if (ctx->Pixel.IndexShift || ctx->Pixel.IndexOffset ||
                  ctx->Pixel.MapStencilFlag) {
               /* make a copy of input */
               memcpy(stencil, source, n * sizeof(GLubyte));
               _mesa_apply_stencil_transfer_ops(ctx, n, stencil);
               source = stencil;
             }

            switch (dstType) {
            case GL_UNSIGNED_BYTE:
               memcpy(dest, source, n);
               break;

                  map += stride;
               }
            ...
   }
}
```

- 该接口同样通过MapRenderbuffer获取内存指针，然后通过_mesa_unpack_ubyte_stencil_row进行像素解包填充

depth

```
/**
 * Read pixels for format=GL_DEPTH_COMPONENT.
 */
static void
read_depth_pixels( struct gl_context *ctx,
                   GLint x, GLint y,
                   GLsizei width, GLsizei height,
                   GLenum type, GLvoid *pixels,
                   const struct gl_pixelstore_attrib *packing )
{
   struct gl_framebuffer *fb = ctx->ReadBuffer;
   struct gl_renderbuffer *rb = fb->Attachment[BUFFER_DEPTH].Renderbuffer;
   GLint j;
   GLubyte *dst, *map;
   int dstStride, stride;
   GLfloat *depthValues;

   if (!rb)
```

```
      return;

 (type == GL_UNSIGNED_INT &&
       read_uint_depth_pixels(ctx, x, y, width, height, type, pixels, packing)) {
      return;
   }

   dstStride = _mesa_image_row_stride(packing, width, GL_DEPTH_COMPONENT, type);
   dst = (GLubyte *) _mesa_image_address2d(packing, pixels, width, height,
                       GL_DEPTH_COMPONENT, type, 0, 0);

   ctx->Driver.MapRenderbuffer(ctx, rb, x, y, width, height, GL_MAP_READ_BIT,
                   &map, &stride, fb->FlipY);
   if (!map) {
      _mesa_error(ctx, GL_OUT_OF_MEMORY, "glReadPixels");
      return;
   }

   depthValues = malloc(width * sizeof(GLfloat));

   if (depthValues) {
      /* General case (slower) */
      for (j = 0; j < height; j++, y++) {
         _mesa_unpack_float_z_row(rb->Format, width, map, depthValues);
                   unpack_float_z_func unpack;

                   switch (format)
                   case MESA_FORMAT_S8_UINT_Z24_UNORM:
                   case MESA_FORMAT_X8_UINT_Z24_UNORM:
                      unpack = unpack_float_z_X8_UINT_Z24_UNORM;
                      break;
                   ....

                   unpack(n, src, dst);



         _mesa_pack_depth_span(ctx, width, dst, type, depthValues, packing);

         dst += dstStride;
         map += stride;
      }
   }
}
```

- 对于深度附件格式，这里采用的是_mesa_unpack_float_z_row 解包z值，
- 在_mesa_unpack_float_z_row内， 首先是根据格式决定不同的unpack接口函数，然后调用 unpack_float_z_func 函数进行计算
- 关于unpack_float_z_func ,定义

```
typedef void (*unpack_float_z_func)(GLuint n, const void *src, GLfloat *dst);
```

- 比如对于 unpack_float_z_X8_UINT_Z24_UNORM

```
static void
unpack_float_z_X8_UINT_Z24_UNORM(GLuint n, const void *src, GLfloat *dst)
{
   /* only return Z, not stencil data */
   const GLuint *s = ((const GLuint *) src);
   const GLdouble scale = 1.0 / (GLdouble) 0xffffff;
   GLuint i;
   for (i = 0; i < n; i++) {
      dst[i] = (GLfloat) ((s[i] >> 8) * scale);
      assert(dst[i] >= 0.0F);
      assert(dst[i] <= 1.0F);
   }
}
```

- 这段代码是一个用于将输入数据（格式为X8_UINT_Z24_UNORM）解包成浮点数（GLfloat）的函数。该格式通常用于深度缓冲区（Z缓冲区）的存储，其中最低8位表示整数部分（X8_UINT），接下来的24位表示小数部分（Z24_UNORM）。解包后的浮点数范围在[0.0, 1.0]之间。
-
- 函数名为`unpack_float_z_X8_UINT_Z24_UNORM`，表明它的作用是解包深度缓冲区的数据。
- 输入参数包括要解包的元素个数`n`，输入数据指针`src`，以及存储解包结果的浮点数数组指针`dst`。
- 该函数使用了一个循环，遍历每个元素，对输入数据进行解包并存储到浮点数数组中。
- 在解包过程中，通过位移操作`(s[i] >> 8)`，获取到 Z24_UNORM 部分的数值，然后乘以缩放因子`scale`，得到浮点数的值。这个缩放因子是为了将Z24_UNORM的整数值映射到[0.0, 1.0]的范围。
- 在每次解包后，使用`assert`断言确保解包得到的值在合理的范围内，即大于等于0.0且小于等于1.0。
-
- 总体来说，这段代码的目的是将深度缓冲区的X8_UINT_Z24_UNORM格式的数据解包成浮点数，并通过缩放操作将其映射到标准的深度范围[0.0, 1.0]。

depth_stencil

```
/**
 * Read combined depth/stencil values.
 * We'll have already done error checking to be sure the expected
 * depth and stencil buffers really exist.
 */
static void
read_depth_stencil_pixels(struct gl_context *ctx,
                          GLint x, GLint y,
                          GLsizei width, GLsizei height,
                          GLenum type, GLvoid *pixels,
                          const struct gl_pixelstore_attrib *packing )
{
```

```
   const GLboolean scaleOrBias
      = ctx->Pixel.DepthScale != 1.0F || ctx->Pixel.DepthBias != 0.0F;
   const GLboolean stencilTransfer = ctx->Pixel.IndexShift
      || ctx->Pixel.IndexOffset || ctx->Pixel.MapStencilFlag;
   GLubyte *dst;
   int dstStride;

   dst = (GLubyte *) _mesa_image_address2d(packing, pixels,)

   dstStride = _mesa_image_row_stride(packing, width,

   /* Fast 24/8 reads. */
   if (type == GL_UNSIGNED_INT_24_8 &&
       !scaleOrBias && !stencilTransfer && !packing->SwapBytes) {
      if (fast_read_depth_stencil_pixels(ctx, x, y, width, height,
                     dst, dstStride))
     return;

      if (fast_read_depth_stencil_pixels_separate(ctx, x, y, width, height,
                          (uint32_t *)dst, dstStride))
     return;
   }

    // 使用缓慢的方法访问深度模板像素分开读取
   slow_read_depth_stencil_pixels_separate(ctx, x, y, width, height,
                       type, packing,
                       dst, dstStride);
}
```

- 对于类型为GL_UNSIGNED_INT_24_8的格式类型存在快速读取像素操作
- 否则通过分开读取像素的操作读取像素

color

亮度格式都是在兼容模式下使用

```
/*
 * 读取R、G、B、A、RGB、L、LA像素。
 */
static void
read_rgba_pixels(struct gl_context *ctx,
                 GLint x, GLint y,
                 GLsizei width, GLsizei height,
                 GLenum format, GLenum type, GLvoid *pixels,
                 const struct gl_pixelstore_attrib *packing)
{
   GLbitfield transferOps;
   bool dst_is_integer, convert_rgb_to_lum, needs_rebase;
   int dst_stride, src_stride, rb_stride;
   uint32_t dst_format, src_format;
   GLubyte *dst, *map;
   mesa_format rb_format;
   bool needs_rgba;
   void *rgba, *src;
```

```
   bool src_is_uint = false;
   uint8_t rebase_swizzle[4];
   struct gl_framebuffer *fb = ctx->ReadBuffer;
   struct gl_renderbuffer *rb = fb->_ColorReadBuffer;
   GLenum dstBaseFormat = _mesa_unpack_format_to_base_format(format);

   if (!rb)
      return;

   transferOps = _mesa_get_readpixels_transfer_ops(ctx, rb->Format, format,
                                                   type, GL_FALSE);
   /* 描述目标格式 */
   dst_is_integer = _mesa_is_enum_format_integer(format);
   dst_stride = _mesa_image_row_stride(packing, width, format, type);
   //  从格式类型转换为内部格式 see format.pdf
   dst_format = _mesa_format_from_format_and_type(format, type);
   convert_rgb_to_lum =
      _mesa_need_rgb_to_luminance_conversion(rb->_BaseFormat, dstBaseFormat);
   dst = (GLubyte *)_mesa_image_address2d(packing, pixels, width, height,
                                          format, type, 0, 0);

   /* 映射源渲染缓冲区 */
   ctx->Driver.MapRenderbuffer(ctx, rb, x, y, width, height, GL_MAP_READ_BIT,
                               &map, &rb_stride, fb->FlipY);
   rb_format = _mesa_get_srgb_format_linear(rb->Format);

   /*
    * 根据转换涉及的基本格式，我们可能需要重新基准化一些值，因此对于这些格式，我们计算一个重新基准化 swizzle。
    */
   if (rb->_BaseFormat == GL_LUMINANCE || rb->_BaseFormat == GL_INTENSITY) {
        ...
   } else if (rb->_BaseFormat == GL_LUMINANCE_ALPHA) {
        ...
   } else if (_mesa_get_format_base_format(rb_format) != rb->_BaseFormat) {
      needs_rebase =
         _mesa_compute_rgba2base2rgba_component_mapping(rb->_BaseFormat,
                                                        rebase_swizzle);
   } else {
      needs_rebase = false;
   }

   /* 由于 _mesa_format_convert 不处理 transferOps，我们需要在调用函数之前处理它们。
    * 这需要首先转换为RGBA浮点数，以便我们可以调用 _mesa_apply_rgba_transfer_ops。
    * 如果目标格式是整数，则 transferOps 不适用。
    *
    * 转换为亮度还需要先转换为RGBA，以便我们可以计算亮度值为L=R+G+B。
    * 请注意，这与 GetTexImage 不同，其中我们计算 L=R。
    */

   needs_rgba = transferOps || convert_rgb_to_lum;
   rgba = NULL;
   if (needs_rgba) {
        ...
   } else {
      /* 不需要RGBA转换，直接转换为目标 */
```

```
      src = map;
      src_format = rb_format;
      src_stride = rb_stride;
   }

   /* 进行转换。
    *
    * 如果目标格式是亮度（Luminance），我们需要通过计算 L=R+G+B 进行转换。
    */
   if (!convert_rgb_to_lum) {
      _mesa_format_convert(dst, dst_format, dst_stride,
                           src, src_format, src_stride,
                           width, height,
                           needs_rebase ? rebase_swizzle : NULL);
   } else if (!dst_is_integer) {
        ...
   } else {
      /* 从RGBA整数值计算整数亮度值 */
        ...
   }

   free(rgba);

done_swap:
   /* 如果需要，处理字节交换 */
   if (packing->SwapBytes) {
      _mesa_swap_bytes_2d_image(format, type, packing,
                                width, height, dst, dst);
   }

done_unmap:
   ctx->Driver.UnmapRenderbuffer(ctx, rb);
}
```

- 该函数通过打包计算公式对像素格式进行打包,实际上就是teximage接口下通过store_rgba上传纹理数的逆转换, see texture_format.pdf

图像复制接口 CopyImageSubData / st_CopyImageSubData

```
// st_cb_copyimage.c

void
st_init_copy_image_functions(struct dd_function_table *functions)
{
   functions->CopyImageSubData = st_CopyImageSubData;
}
```

图像复制像素 CopyImageSubData /st_CopyImageSubData

```
static void
st_CopyImageSubData(struct gl_context *ctx,
                    struct gl_texture_image *src_image,
                    struct gl_renderbuffer *src_renderbuffer,
```

```
                    int src_x, int src_y, int src_z,
                    struct gl_texture_image *dst_image,
                    struct gl_renderbuffer *dst_renderbuffer,
                    int dst_x, int dst_y, int dst_z,
                    int src_width, int src_height)
{
   struct st_context *st = st_context(ctx);
   struct pipe_context *pipe = st->pipe;
   struct pipe_resource *src_res, *dst_res;
   struct pipe_box box;
   int src_level, dst_level;
   int orig_src_z = src_z, orig_dst_z = dst_z;

   st_flush_bitmap_cache(st);
   st_invalidate_readpix_cache(st);

   if (src_image) {
    // 处理纹理
      struct st_texture_image *src = st_texture_image(src_image);
      src_res = src->pt;
      src_level = src_image->Level;
      src_z += src_image->Face;
      if (src_image->TexObject->Immutable) {
         src_level += src_image->TexObject->MinLevel;
         src_z += src_image->TexObject->MinLayer;
      }
   } else {
   // 处理renderbuffer绑定
      struct st_renderbuffer *src = st_renderbuffer(src_renderbuffer);
      src_res = src->texture;
      src_level = 0;
   }

   if (dst_image) {
      struct st_texture_image *dst = st_texture_image(dst_image);
      dst_res = dst->pt;
      dst_level = dst_image->Level;
      dst_z += dst_image->Face;
      if (dst_image->TexObject->Immutable) {
         dst_level += dst_image->TexObject->MinLevel;
         dst_z += dst_image->TexObject->MinLayer;
      }
   } else {
      struct st_renderbuffer *dst = st_renderbuffer(dst_renderbuffer);
      dst_res = dst->texture;
      dst_level = 0;
   }

   u_box_2d_zslice(src_x, src_y, src_z, src_width, src_height, &box);

   if ((src_image && st_compressed_format_fallback(st, src_image->TexFormat)) ||
       (dst_image && st_compressed_format_fallback(st, dst_image->TexFormat))) {
       // 如果驱动不支持的et1,et2, astc_2d 压缩格式图像通过直接映射拷贝
      fallback_copy_image(st, dst_image, dst_res, dst_x, dst_y, orig_dst_z,
                          src_image, src_res, src_x, src_y, orig_src_z,
                          src_width, src_height);
```

```
   } else {
      copy_image(pipe, dst_res, dst_level, dst_x, dst_y, dst_z,
                 src_res, src_level, &box);
   }
}
```

- 内部通过两种方式拷贝数据， 一是通过blit, 二是通过Map

使用blit 复制图像数据

```
// 复制图像数据的函数

static void
copy_image(struct pipe_context *pipe,
           struct pipe_resource *dst,
           unsigned dst_level,
           unsigned dstx, unsigned dsty, unsigned dstz,
           struct pipe_resource *src,
           unsigned src_level,
           const struct pipe_box *src_box)
{
   // 如果源和目标格式相同，或者源/目标格式是压缩格式，则直接调用底层的资源复制函数
   if (src->format == dst->format ||
       util_format_is_compressed(src->format) ||
       util_format_is_compressed(dst->format)) {
      pipe->resource_copy_region(pipe, dst, dst_level, dstx, dsty, dstz,
                                 src, src_level, src_box);
            [jump radeonsi si_resource_copy_region]
      return;
   }

   /* 对于B10G10R10*2格式，需要使用R10G10B10A2格式作为临时纹理进行两次拷贝。
    */
   if (handle_complex_copy(pipe, dst, dst_level, dstx, dsty, dstz, src,
                           src_level, src_box, PIPE_FORMAT_B10G10R10A2_UINT,
                           PIPE_FORMAT_R10G10B10A2_UINT))
      ...
      blit(pipe, temp, canon_format, 0, 0, 0, 0, src, noncanon_format,
           src_level, src_box);
      swizzled_copy(pipe, dst, dst_level, dstx, dsty, dstz, temp, 0,
                    &temp_box);

      return;

   /* 对于G8R8格式，需要使用R8G8格式作为临时纹理进行两次拷贝。
    */
   if (handle_complex_copy(pipe, dst, dst_level, dstx, dsty, dstz, src,
                           src_level, src_box, PIPE_FORMAT_G8R8_UNORM,
                           PIPE_FORMAT_R8G8_UNORM))
      return;

   /* 对于G16R16格式，需要使用R16G16格式作为临时纹理进行两次拷贝。
    */
```

```
   if (handle_complex_copy(pipe, dst, dst_level, dstx, dsty, dstz, src,
                           src_level, src_box, PIPE_FORMAT_G16R16_UNORM,
                           PIPE_FORMAT_R16G16_UNORM))
      return;

   /* 只允许在RGBA8格式上进行非标识置换。 */

   /* 简单拷贝，使用置换进行内存拷贝，无格式转换。*/
   swizzled_copy(pipe, dst, dst_level, dstx, dsty, dstz, src, src_level,
                 src_box);
        ...
        blit(pipe, dst, blit_dst_format, dst_level, dstx, dsty, dstz,
             src, blit_src_format, src_level, src_box);
            pipe->blit(pipe, &blit);
                [jump RadeonSI si_blit]

}
```

通过Map(MapTextureImage, pipe_transfer_map)复制

```
static void
fallback_copy_image(struct st_context *st,
                    struct gl_texture_image *dst_image,
                    struct pipe_resource *dst_res,
                    int dst_x, int dst_y, int dst_z,
                    struct gl_texture_image *src_image,
                    struct pipe_resource *src_res,
                    int src_x, int src_y, int src_z,
                    int src_w, int src_h)
{
   uint8_t *dst, *src;
   int dst_stride, src_stride;
   struct pipe_transfer *dst_transfer, *src_transfer;
   unsigned line_bytes;

   bool dst_is_compressed = dst_image && _mesa_is_format_compressed(dst_image-
>TexFormat);
   bool src_is_compressed = src_image && _mesa_is_format_compressed(src_image-
>TexFormat);

   unsigned dst_blk_w = 1, dst_blk_h = 1, src_blk_w = 1, src_blk_h = 1;
   if (dst_image)
      _mesa_get_format_block_size(dst_image->TexFormat, &dst_blk_w, &dst_blk_h);
   if (src_image)
      _mesa_get_format_block_size(src_image->TexFormat, &src_blk_w, &src_blk_h);

   unsigned dst_w = src_w;
   unsigned dst_h = src_h;
   unsigned lines = src_h;

   if (src_is_compressed && !dst_is_compressed) {
      dst_w = DIV_ROUND_UP(dst_w, src_blk_w);
      dst_h = DIV_ROUND_UP(dst_h, src_blk_h);
   } else if (!src_is_compressed && dst_is_compressed) {
      dst_w *= dst_blk_w;
```

```
      dst_h *= dst_blk_h;
   }
   if (src_is_compressed) {
      lines = DIV_ROUND_UP(lines, src_blk_h);
   }

   if (src_image)
      line_bytes = _mesa_format_row_stride(src_image->TexFormat, src_w);
   else
      line_bytes = _mesa_format_row_stride(dst_image->TexFormat, dst_w);

   if (dst_image) {
      st->ctx->Driver.MapTextureImage(
            st->ctx, dst_image, dst_z,
            dst_x, dst_y, dst_w, dst_h,
            GL_MAP_WRITE_BIT, &dst, &dst_stride);
   } else {
      dst = pipe_transfer_map(st->pipe, dst_res, 0, dst_z,
                              PIPE_TRANSFER_WRITE,
                              dst_x, dst_y, dst_w, dst_h,
                              &dst_transfer);
      dst_stride = dst_transfer->stride;
   }

   if (src_image) {
      st->ctx->Driver.MapTextureImage(
            st->ctx, src_image, src_z,
            src_x, src_y, src_w, src_h,
            GL_MAP_READ_BIT, &src, &src_stride);
   } else {
      src = pipe_transfer_map(st->pipe, src_res, 0, src_z,
                              PIPE_TRANSFER_READ,
                              src_x, src_y, src_w, src_h,
                              &src_transfer);
      src_stride = src_transfer->stride;
   }

   for (int y = 0; y < lines; y++) {
      memcpy(dst, src, line_bytes);
      dst += dst_stride;
      src += src_stride;
   }

    ...
}
```

radeonsi

blit (Block Image Transfer)接口

```
/**
 * 用于类似拷贝功能的资源函数
 *
 * 如果驱动程序支持多采样，blit函数必须实现颜色解析。
 */
/*@{*/

/**
 * 从一个资源复制一块像素到另一个资源。
 * 这两个资源必须具有相同的格式。
 * 不允许对具有nr_samples > 1的资源进行此操作。
 */
void (*resource_copy_region)(struct pipe_context *pipe,
                             struct pipe_resource *dst,
                             unsigned dst_level,
                             unsigned dstx, unsigned dsty, unsigned dstz,
                             struct pipe_resource *src,
                             unsigned src_level,
                             const struct pipe_box *src_box);

/* 用于像素块拷贝的最优硬件路径。
 * 允许缩放、格式转换、上采样和下采样（解析）。
 */
void (*blit)(struct pipe_context *pipe,
             const struct pipe_blit_info *info);


// si_blit.c
void si_init_blit_functions(struct si_context *sctx)
{
    sctx->b.resource_copy_region = si_resource_copy_region;
    sctx->b.blit = si_blit;
    sctx->b.flush_resource = si_flush_resource;
    sctx->b.generate_mipmap = si_generate_mipmap;
}
```

si_blit

```

static void si_blit(struct pipe_context *ctx,
            const struct pipe_blit_info *info)
{
    struct si_context *sctx = (struct si_context*)ctx;
    struct si_texture *dst = (struct si_texture *)info->dst.resource;

    if (do_hardware_msaa_resolve(ctx, info)) {
        return;
    }
    // 分支,使用sdma dma_copy,see dma_cs.pdf
```

```
    ... // dcc等处理

    si_blitter_begin(sctx, SI_BLIT |
            (info->render_condition_enable ? 0 : SI_DISABLE_RENDER_COND));
    util_blitter_blit(sctx->blitter, info);
    si_blitter_end(sctx);
}
```

- si_blit 两种路径，一是通过sdma copy 线性纹理， 二是通过util_blitter_blit 拷贝， see blitter.pdf

si_resource_copy_region

```
void si_resource_copy_region(struct pipe_context *ctx,
                struct pipe_resource *dst,
                unsigned dst_level,
                unsigned dstx, unsigned dsty, unsigned dstz,
                struct pipe_resource *src,
                unsigned src_level,
                const struct pipe_box *src_box)
{
    struct si_context *sctx = (struct si_context *)ctx;
    struct si_texture *ssrc = (struct si_texture*)src;
    struct pipe_surface *dst_view, dst_templ;
    struct pipe_sampler_view src_templ, *src_view;
    unsigned dst_width, dst_height, src_width0, src_height0;
    unsigned dst_width0, dst_height0, src_force_level = 0;
    struct pipe_box sbox, dstbox;

    /* Handle buffers first. */
    if (dst->target == PIPE_BUFFER && src->target == PIPE_BUFFER) {
        si_copy_buffer(sctx, dst, src, dstx, src_box->x, src_box->width);
        return;
    }
    ...

    dst_width = u_minify(dst->width0, dst_level);
    dst_height = u_minify(dst->height0, dst_level);
    dst_width0 = dst->width0;
    dst_height0 = dst->height0;
    src_width0 = src->width0;
    src_height0 = src->height0;

    util_blitter_default_dst_texture(&dst_templ, dst, dst_level, dstz);
    util_blitter_default_src_texture(sctx->blitter, &src_templ, src, src_level);

    src_templ.format = ...;
    dst_templ.format = ...;
    ...

    dstx = ...;

    /* Initialize the surface. */
    dst_view = si_create_surface_custom(ctx, dst, &dst_templ,
                        dst_width0, dst_height0,
```

```
                          dst_width, dst_height);

    /* Initialize the sampler view. */
    src_view = si_create_sampler_view_custom(ctx, src, &src_templ,
                         src_width0, src_height0,
                         src_force_level);

    u_box_3d(dstx, dsty, dstz, abs(src_box->width), abs(src_box->height),
         abs(src_box->depth), &dstbox);

    /* Copy. */
    si_blitter_begin(sctx, SI_COPY);
    util_blitter_blit_generic(sctx->blitter, dst_view, &dstbox,
                  src_view, src_box, src_width0, src_height0,
                  PIPE_MASK_RGBAZS, PIPE_TEX_FILTER_NEAREST, NULL,
                  false);
    si_blitter_end(sctx);

    pipe_surface_reference(&dst_view, NULL);
    pipe_sampler_view_reference(&src_view, NULL);
}
```

buffers的资源拷贝

```
void si_copy_buffer(struct si_context *sctx,
            struct pipe_resource *dst, struct pipe_resource *src,
            uint64_t dst_offset, uint64_t src_offset, unsigned size)
{
    if (!size)
        return;

    enum si_coherency coher = SI_COHERENCY_SHADER;
    enum si_cache_policy cache_policy = get_cache_policy(sctx, coher, size);

    /* Only use compute for VRAM copies on dGPUs. */
    if (sctx->screen->info.has_dedicated_vram &&
        r600_resource(dst)->domains & RADEON_DOMAIN_VRAM &&
        r600_resource(src)->domains & RADEON_DOMAIN_VRAM &&
        size > 32 * 1024 &&
        dst_offset % 4 == 0 && src_offset % 4 == 0 && size % 4 == 0) {
        si_compute_do_clear_or_copy(sctx, dst, dst_offset, src, src_offset,
                        size, NULL, 0, coher);
    } else {
        si_cp_dma_copy_buffer(sctx, dst, src, dst_offset, src_offset, size,
                      0, coher, cache_policy);
    }
}
```

- 采用compute shader或者 cp dma 进行拷贝数据, see compute_shader.pdf, cp_dma.pdf

texture的资源拷贝

使用util_blitter_blit_generic 将源/目的buffer通过创建pipe_sampler_view 绑定在fs shader中,再通过draw_rectangle完成拷贝
see blitter.pdf

flush_resource

see flush.pdf

si_generate_mipmap

see texture_mipmap.pdf

调试

使用piglit 的fbo-readpixels
根据transfer_map 日志可跟踪像素数据

像素计算的实用接口

计算像素的图像地址

```
/**
 * 返回图像中特定像素的地址（1D、2D或3D）。
 *
 * 根据 \p packing 观察像素的解包/打包参数。
 *
 * \param dimensions 表示图像的维度，可以是1、2或3
 * \param packing 像素存储属性
 * \param image 图像数据的起始地址
 * \param width 图像宽度
 * \param height 图像高度
 * \param format 像素格式（必须事先验证）
 * \param type 像素数据类型（必须事先验证）
 * \param img 体积中的哪个图像（对于1D或2D图像为0）
 * \param row 图像中的像素行（对于1D图像为0）
 * \param column 图像中的像素列
 *
 * \return 像素的地址。
 *
 * \sa gl_pixelstore_attrib。
 */
GLvoid *
_mesa_image_address(GLuint dimensions,
                    const struct gl_pixelstore_attrib *packing,
                    const GLvoid *image,
                    GLsizei width, GLsizei height,
                    GLenum format, GLenum type,
                    GLint img, GLint row, GLint column)
{
   const GLubyte *addr = (const GLubyte *)image;

   addr += _mesa_image_offset(dimensions, packing, width, height,
```

```
                                format, type, img, row, column);

   return (GLvoid *)addr;
}
```

计算图像行之间的步幅

```
/**
 * 计算图像行之间的步幅（以字节为单位）。
 *
 * \param packing 像素存储属性
 * \param width 图像宽度。
 * \param format 像素格式。
 * \param type 像素数据类型。
 *
 * \return 给定参数的字节步幅，如果有错误则返回-1
 */
GLint
_mesa_image_row_stride(const struct gl_pixelstore_attrib *packing,
                       GLint width, GLenum format, GLenum type)
{
   GLint bytesPerRow, remainder;

   assert(packing);

   if (type == GL_BITMAP) {
      if (packing->RowLength == 0) {
         bytesPerRow = (width + 7) / 8;
      }
      else {
         bytesPerRow = (packing->RowLength + 7) / 8;
      }
   }
   else {
      /* 非位图数据 */
      const GLint bytesPerPixel = _mesa_bytes_per_pixel(format, type);
      if (bytesPerPixel <= 0)
         return -1;  /* 错误 */
      if (packing->RowLength == 0) {
         bytesPerRow = bytesPerPixel * width;
      }
      else {
         bytesPerRow = bytesPerPixel * packing->RowLength;
      }
   }

   remainder = bytesPerRow % packing->Alignment;
   if (remainder > 0) {
      bytesPerRow += (packing->Alignment - remainder);
   }

   if (packing->Invert) {
      /* 对字节每行取负值（负的行步幅） */
      bytesPerRow = -bytesPerRow;
   }
```

```
   return bytesPerRow;
}
```

计算图像特定像素的字节偏移量

```
/**
 * 返回图像中特定像素的字节偏移量（1D、2D或3D）。
 *
 * 根据 \p packing 观察像素的解包/打包参数。
 *
 * \param dimensions 表示图像的维度，可以是1、2或3
 * \param packing 像素存储属性
 * \param width 图像宽度
 * \param height 图像高度
 * \param format 像素格式（必须事先验证）
 * \param type 像素数据类型（必须事先验证）
 * \param img 体积中的哪个图像（对于1D或2D图像为0）
 * \param row 图像中的像素行（对于1D图像为0）
 * \param column 图像中的像素列
 *
 * \return 像素的偏移量。
 *
 * \sa gl_pixelstore_attrib。
 */
GLintptr
_mesa_image_offset(GLuint dimensions,
                   const struct gl_pixelstore_attrib *packing,
                   GLsizei width, GLsizei height,
                   GLenum format, GLenum type,
                   GLint img, GLint row, GLint column)
{
   GLint alignment;        /* 1, 2,或4 */
   GLint pixels_per_row;
   GLint rows_per_image;
   GLint skiprows;
   GLint skippixels;
   GLint skipimages;       /* 用于3D体积图像 */
   GLintptr offset;

   assert(dimensions >= 1 && dimensions <= 3);

   alignment = packing->Alignment;
   if (packing->RowLength > 0) {
      pixels_per_row = packing->RowLength;
   }
   else {
      pixels_per_row = width;
   }
   if (packing->ImageHeight > 0) {
      rows_per_image = packing->ImageHeight;
   }
   else {
      rows_per_image = height;
   }
```

```
   skippixels = packing->SkipPixels;
   /* 注意：对于1D图像，_使用_ SKIP_ROWS */
   skiprows = packing->SkipRows;
   /* 注意：SKIP_IMAGES仅用于3D图像 */
   skipimages = (dimensions == 3) ? packing->SkipImages : 0;

   if (type == GL_BITMAP) {
      /* 位图数据 */
      GLintptr bytes_per_row;
      GLintptr bytes_per_image;
      /* 颜色或模板索引每像素的分量： */
      const GLint comp_per_pixel = 1;

      /* 像素类型和格式应该在之前已经经过错误检查 */
      assert(format == GL_COLOR_INDEX || format == GL_STENCIL_INDEX);

      bytes_per_row = alignment
                    * DIV_ROUND_UP(comp_per_pixel * pixels_per_row, 8 *
alignment);

      bytes_per_image = bytes_per_row * rows_per_image;

      offset = (skipimages + img) * bytes_per_image
               + (skiprows + row) * bytes_per_row
               + (skippixels + column) / 8;
   }
   else {
      /* 非位图数据 */
      GLintptr bytes_per_pixel, bytes_per_row, remainder, bytes_per_image;
      GLintptr topOfImage;

      bytes_per_pixel = _mesa_bytes_per_pixel(format, type);

      /* 像素类型和格式应该在之前已经经过错误检查 */
      assert(bytes_per_pixel > 0);

      bytes_per_row = pixels_per_row * bytes_per_pixel;
      remainder = bytes_per_row % alignment;
      if (remainder > 0)
         bytes_per_row += (alignment - remainder);

      assert(bytes_per_row % alignment == 0);

      bytes_per_image = bytes_per_row * rows_per_image;

      if (packing->Invert) {
         /* 将像素_addr设置为最后一行 */
         topOfImage = bytes_per_row * (height - 1);
         bytes_per_row = -bytes_per_row;
      }
      else {
         topOfImage = 0;
      }

      /* 计算最终像素地址 */
```

```
      offset = (skipimages + img) * bytes_per_image
               + topOfImage
               + (skiprows + row) * bytes_per_row
               + (skippixels + column) * bytes_per_pixel;
   }

   return offset;
}
```

计算每个像素的字节数

```
/**
 * Get the bytes per pixel of pixel format type pair.
 *
 * \param format pixel format.
 * \param type pixel type.
 *
 * \return bytes per pixel, or -1 if a bad format or type was given.
 */
GLint
_mesa_bytes_per_pixel(GLenum format, GLenum type)
{
    // 获取像素的格式分量数
   GLint comps = _mesa_components_in_format(format);
   if (comps < 0)
      return -1;

   switch (type) {
   case GL_BITMAP:
      return 0;  /* special case */
   case GL_BYTE:
   case GL_UNSIGNED_BYTE:
      return comps * sizeof(GLubyte);
   case GL_SHORT:
   case GL_UNSIGNED_SHORT:
      return comps * sizeof(GLshort);
   ...
      return -1;
   }
}
```

- 这里的类型会出现GL_BITMAP 类型，默认返回0

计算像素格式分量数

```
_mesa_components_in_format(GLenum format)
   switch (format) {
   case GL_COLOR_INDEX:
      return 1;

   ...
   case GL_RG_INTEGER:
```

```
      return 2;

   ...
   case GL_RGB_INTEGER_EXT:
   case GL_BGR_INTEGER_EXT:
      return 3;

   case GL_RGBA:
   ...
   case GL_BGRA_INTEGER_EXT:
      return 4;

   default:
      return -1;
   }
}
```