fb

9.1 帧缓冲概述

帧缓冲由一组像素组成，排列成二维数组。在本讨论中，帧缓冲中的每个像素仅仅是一组某个数量的位。每个像素的位数可能会根据GL实现、选择的帧缓冲类型以及在创建帧缓冲时指定的参数而变化。默认帧缓冲的创建和管理超出了本规范的范围，而帧缓冲对象的创建和管理在第9.2节中有详细描述。

帧缓冲中的每个像素的对应位被组合到一起形成一个位平面；每个位平面包含来自每个像素的一个单独的位。这些位平面分为几个逻辑缓冲区。这些缓冲区是颜色、深度和模板缓冲区。颜色缓冲实际上由许多缓冲区组成，这些颜色缓冲区根据GL是否绑定到默认帧缓冲或帧缓冲对象，具有相关但略有不同的用途。

对于默认帧缓冲，颜色缓冲区分别是前左缓冲区、前右缓冲区、后左缓冲区和后右缓冲区。通常，前缓冲区的内容显示在彩色监视器上，而后缓冲区的内容是不可见的（单视觉上下文只显示前左缓冲区；立体视觉上下文同时显示前左和前右缓冲区）。所有颜色缓冲区必须具有相同数量的位平面，尽管实现或上下文可能选择不提供右缓冲区，或者根本不提供后缓冲区。此外，实现或上下文可能选择不提供深度或模板缓冲区。

如果没有默认帧缓冲与GL上下文关联，则帧缓冲是不完整的，除非绑定了帧缓冲对象（请参见第9.2节和9.4节）。帧缓冲对象不可见，并且不具有默认帧缓冲中存在的任何颜色缓冲区。相反，帧缓冲对象的缓冲区是通过将单个纹理或渲染缓冲（请参见第9节）附加到一组附着点来指定的。帧缓冲对象具有一系列颜色缓冲区附着点，编号从零到n，一个深度缓冲区附着点和一个模板缓冲区附着点。为了用于渲染，帧缓冲对象必须是完整的，如第9.4节所述。不需要填充帧缓冲对象的所有附着。

颜色缓冲中的每个像素由最多四个颜色分量组成。这四个颜色分量分别命名为R、G、B和A；颜色缓冲区不要求具有所有四个颜色分量。R、G、B和A分量可以表示为有符号或无符号的规格化定点、浮点、有符号或无符号的整数值；所有分量必须具有相同的表示形式。

深度缓冲区中的每个像素由一个描述在第13.6.1节中的格式中的单个无符号整数值或浮点值组成。模板缓冲区中的每个像素由一个单个无符号整数值组成。

颜色、深度和模板缓冲区中的位平面数取决于当前绑定的帧缓冲。对于默认帧缓冲，位平面的数量是固定的。对于帧缓冲对象，给定逻辑缓冲区中的位平

面数量可能会在附加到相应附着点的图像发生变化时发生变化。

GL有两个活动的帧缓冲；绘制帧缓冲是渲染操作的目标，读取帧缓冲是读取操作的源。同一帧缓冲可以同时用于绘制和读取。第9.2节描述了控制帧缓冲使用的机制。

默认帧缓冲最初用作绘制和读取帧缓冲1，所有提供的位平面的初始状态是未定义的。可以通过查询来获取绘制和读取帧缓冲中缓冲区的格式和编码，具体方法请参见第9.2.3节。

9.2 绑定和管理帧缓冲对象

帧缓冲对象封装了帧缓冲的状态，类似于纹理对象封装了纹理的状态。具体而言，帧缓冲对象封装了描述一组颜色、深度和模板逻辑缓冲所需的状态（不允许其他类型的缓冲）。对于每个逻辑缓冲，可以将帧缓冲可附加的图像附加到帧缓冲中，以存储该逻辑缓冲的渲染输出。帧缓冲可附加的图像示例包括纹理图像和渲染缓冲图像。渲染缓冲的详细描述请参见第9.2.4节。

通过允许将渲染缓冲的图像附加到帧缓冲，GL提供了支持离屏渲染的机制。此外，通过允许将纹理的图像附加到帧缓冲，GL提供了支持渲染到纹理的机制。

用于渲染和读取操作的默认帧缓冲由窗口系统提供。此外，可以创建和操作具有名称的帧缓冲对象。帧缓冲对象的名称空间是无符号整数，其中零由GL保留供默认帧缓冲使用。

通过将GenFramebuffers返回的名称绑定到DRAW_FRAMEBUFFER或READ_FRAMEBUFFER，可以创建帧缓冲对象。通过调用：

```
void BindFramebuffer( enum target, uint framebuffer );
```

其中目标设置为所需的帧缓冲目标，framebuffer设置为帧缓冲对象的名称，可以实现绑定。生成的帧缓冲对象是一个新的状态向量，包括在表23.24中列出的所有状态和相同的初始值，以及每个附着点的表23.25中列出的一组状态值，具有相同的初始值。其中包括MAX_COLOR_ATTACHMENTS个颜色附着点，以及一个深度和一个模板附着点。

`BindFramebuffer` 也可以用于将现有的帧缓冲对象绑定到 `DRAW_FRAMEBUFFER` 和/或 `READ_FRAMEBUFFER`。如果绑定成功，则新绑定的帧缓冲对象的状态不会发生变化，并且任何先前与目标的绑定都会被中断。

如果将帧缓冲对象绑定到 `DRAW_FRAMEBUFFER` 或 `READ_FRAMEBUFFER`，它将分别成为渲染或读取操作的目标，直到它被删除或另一个帧缓冲对象绑定到相应的绑定点。调用 `BindFramebuffer` 时，将目标设置为 `FRAMEBUFFER`，将 `framebuffer` 绑定到绘制和读取目标。

```
void BindFramebuffer( enum target, uint framebuffer );
```

在这里，`target` 被设置为所需的帧缓冲目标，`framebuffer` 被设置为帧缓冲对象的名称。

```
void CreateFramebuffers( sizei n, uint *framebuffers );
```

`CreateFramebuffers` 命令用于创建 `n` 个之前未使用的帧缓冲对象名称，并将它们存储在 `framebuffers` 中。每个对象代表一个新的帧缓冲对象，它是一个状态向量，包含表23.24中列出的所有状态以及表23.25中为帧缓冲对象的每个附着点列出的状态值集，具有相同的初始值。

`DRAW_FRAMEBUFFER` 和 `READ_FRAMEBUFFER` 的初始状态指的是默认帧缓冲。为了不丢失对默认帧缓冲的访问，将其视为一个名称为零的帧缓冲对象。因此，当零绑定到相应的目标时，将渲染到默认帧缓冲并从中读取。在某些实现中，默认帧缓冲的属性可能随时间变化（例如，响应于窗口系统事件，如将上下文附加到新的窗口系统可绘制对象）。

帧缓冲对象（具有非零名称的对象）与默认帧缓冲在几个重要方面有所不同。首先，并且最重要的是，与默认帧缓冲不同，帧缓冲对象具有每个帧缓冲的逻辑缓冲区的可修改附着点。可以将帧缓冲可附着图像附加到这些附着点，并从中分离，这些附着点在第9.2.2节中有进一步描述。

此外，附加到帧缓冲对象的图像的大小和格式完全由GL界面控制，并且不受窗口系统事件的影响，例如像素格式选择、窗口调整大小和显示模式更改。

另外，当渲染到应用程序创建的帧缓冲对象或从中读取时：

- 像素所有权测试始终成功。换句话说，帧缓冲对象拥有其所有像素。
- 没有可见的颜色缓冲位面。这意味着没有对应于背面、前面、左侧或右侧颜色位面的颜色缓冲。
- 唯一的颜色缓冲位面是由名为 `COLOR_ATTACHMENT0` 到 `COLOR_ATTACHMENTn` 的帧缓冲附着点定义的位面。每个 `COLOR_ATTACHMENTi` 遵循 `COLOR_ATTACHMENTi = COLOR_ATTACHMENT0 + i * 2`。
- 唯一的深度缓冲位面是由帧缓冲附着点 `DEPTH_ATTACHMENT` 定义的位面。

- 唯一的模板缓冲位面是由帧缓冲附着点 `STENCIL_ATTACHMENT` 定义的位面。
- 如果附着的大小不完全相同，则渲染的结果仅在适合所有附件的最大区域内定义。此区域定义为每个附件具有左下角为 `(0, 0)` 和右上角为 `(width, height)` 的矩形的交集。执行渲染命令后，此区域之外附件的内容是未定义的（如第2.4节中定义）。

如果没有附件，渲染将受限于一个矩形，其左下角为 `(0, 0)`，右上角为 `(width, height)`，其中 `width` 和 `height` 分别为帧缓冲对象的默认宽度和高度。

- 如果每个附件的层数不完全相同，则渲染将受限于任何附件的最小层数。如果没有附件，则层数将从帧缓冲对象的默认层计数中获取。

命令 `GenFramebuffers` 在数组 `framebuffers` 中返回 `n` 个先前未使用的帧缓冲对象名称。这些名称仅在 `GenFramebuffers` 的目的中被标记为已使用，但只有在首次绑定时才会获取状态和类型。

通过调用 `DeleteFramebuffers` 函数可以删除帧缓冲对象：

```
void DeleteFramebuffers(sizei n, const uint *framebuffers);
```

其中，`framebuffers` 数组包含了需要删除的 `n` 个帧缓冲对象的名称。删除后，帧缓冲对象将不再具有附件，并且其名称将再次变为未使用状态。如果删除了当前绑定到一个或多个目标 `DRAW_FRAMEBUFFER` 或 `READ_FRAMEBUFFER` 的帧缓冲对象，则相当于执行了 `BindFramebuffer` 函数，将相应的目标绑定到了帧缓冲对象零。已经被标记为在 `GenFramebuffers` 中使用的未使用名称将再次被标记为未使用。在 `framebuffers` 中未使用的名称将被静默忽略，零值也会被忽略。

`IsFramebuffer` 函数用于检查给定的名称是否为帧缓冲对象的名称：

```
boolean IsFramebuffer(uint framebuffer);
```

如果 `framebuffer` 是帧缓冲对象的名称，则返回 `TRUE`。如果 `framebuffer` 是零，或者如果 `framebuffer` 是一个非零值，但不是帧缓冲对象的名称，则 `IsFramebuffer` 返回 `FALSE`。

要查询绑定到绘制和读取帧缓冲的名称，可以使用 `GetIntegerv` 函数和相应的参数 `DRAW_FRAMEBUFFER_BINDING` 和 `READ_FRAMEBUFFER_BINDING`：

```
GLint drawFramebuffer, readFramebuffer;
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &drawFramebuffer);
glGetIntegerv(GL_READ_FRAMEBUFFER_BINDING, &readFramebuffer);
```

在这里，`drawFramebuffer` 将包含当前绑定到绘制帧缓冲的名称，而 `readFramebuffer` 将包含当前绑定到读取帧缓冲的名称。`GL_FRAMEBUFFER_BINDING` 也等同于 `GL_DRAW_FRAMEBUFFER_BINDING`，可以根据需要使用。

9.2.1 帧缓冲对象参数

使用以下命令设置帧缓冲对象的参数：

```
void FramebufferParameteri( enum target, enum pname, int param );
void NamedFramebufferParameteri( uint framebuffer, enum pname, int param );
```

对于 `FramebufferParameteri`：

- 该函数将帧缓冲对象绑定到 `target`，`target` 必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。
- 当 `pname` 为 `FRAMEBUFFER_DEFAULT_WIDTH`、`FRAMEBUFFER_DEFAULT_HEIGHT`、`FRAMEBUFFER_DEFAULT_LAYERS`、`FRAMEBUFFER_DEFAULT_SAMPLES` 或 `FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS` 时，`param` 分别表示帧缓冲对象没有附件时的宽度、高度、层数、采样数或采样位置模式。

对于 `NamedFramebufferParameteri`：

- 该函数接受一个 `framebuffer` 参数，表示帧缓冲对象的名称。
- 其他参数和作用与 `FramebufferParameteri` 相同，但作用于指定名称的帧缓冲对象。

当帧缓冲对象具有一个或多个附件时，帧缓冲对象的宽度、高度、层数、采样数和采样位置模式将从附件的属性中派生。当帧缓冲对象没有附件时，这些属性将从帧缓冲对象的参数中获取。当 `pname` 为 `FRAMEBUFFER_DEFAULT_WIDTH`、`FRAMEBUFFER_DEFAULT_HEIGHT`、`FRAMEBUFFER_DEFAULT_LAYERS`、`FRAMEBUFFER_DEFAULT_SAMPLES` 或 `FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS` 时，`param` 将指定帧缓冲对象没有附件时使用的宽度、高度、层数、采样数或采样位置模式。

当帧缓冲对象没有附件时，仅当 `FRAMEBUFFER_DEFAULT_LAYERS` 的值非零时，才将其视为分层结构。仅当 `FRAMEBUFFER_DEFAULT_SAMPLES` 的值非零时，才被视为具有样本缓冲。帧缓冲对象的样本数将以类似于 `RenderbufferStorageMultisample` 命令的方式从 `FRAMEBUFFER_DEFAULT_SAMPLES` 的值中派生，如果帧缓冲对象具有样本缓冲，并且 `FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS` 的值非零，则将其视为具有固定的样本位置模式，类似于 `TexImage2DMultisample`（请参阅第8.8节）。k

9.2.2 将图像附加到帧缓冲对象

可附加到帧缓冲对象的图像可以被附加到对象上或从对象上分离。相反，GL不能更改默认帧缓冲的图像附件。

单个帧缓冲对象可附加到多个帧缓冲对象上，这可能避免了一些数据拷贝，可能降低了内存消耗。

对于每个逻辑缓冲区，帧缓冲对象存储一组状态，该状态定义了逻辑缓冲区的附加点。附加点状态包含足够的信息来标识附加到附加点的单个图像，或者指示没有附加图像。每个逻辑缓冲区附加点状态列在表23.25中。

有几种类型的帧缓冲对象可附加的图像：

- 渲染缓冲对象的图像，其始终是二维的。
- 一维纹理的单个级别，它被视为高度为1的二维图像。
  • 二维、二维多重采样或矩形纹理的单个级别。
  • 立方图纹理级别的单个面，被视为二维图像。
  • 一维或二维数组纹理、二维多重采样数组纹理或三维纹理的单个层，被视为二维图像。
  • 立方图数组纹理的单个层面，被视为二维图像。
  此外，可以将整个三维、立方图、立方图数组或一维或二维数组纹理的整个级别附加到一个附加点。这样的附加被视为一系列排列在层中的二维图像，并且相应的附加点被视为分层的（另见第9.8节）。

9.2.3 帧缓冲对象查询

可以使用以下命令查询帧缓冲对象的参数：

```
void GetFramebufferParameteriv(enum target, enum pname, int *params);
void GetNamedFramebufferParameteriv(uint framebuffer, enum pname, int *params);
```

对于 `GetFramebufferParameteriv`，帧缓冲对象是绑定到 `target` 的对象，`target` 必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。`FRAMEBUFFER` 等效于 `DRAW_FRAMEBUFFER`。对于 `GetNamedFramebufferParameteriv`，`framebuffer` 可以是零，表示默认的绘制帧缓冲，或者是帧缓冲对象的名称。

`pname` 可能是 `FRAMEBUFFER_DEFAULT_WIDTH`、`FRAMEBUFFER_DEFAULT_HEIGHT`、`FRAMEBUFFER_DEFAULT_LAYERS`、`FRAMEBUFFER_DEFAULT_SAMPLES` 或 `FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS` 中的一个，表示使用 `FramebufferParameteri`（参见第9.2.1节）设置的相应参数之一。这些值只能从帧缓冲对象查询，而不能从默认帧缓冲查询。

`pname` 也可能是 `DOUBLEBUFFER`、`IMPLEMENTATION_COLOR_READ_FORMAT`、`IMPLEMENTATION_COLOR_READ_TYPE`、`SAMPLES`、`SAMPLE_BUFFERS` 或 `STEREO` 中的一个，表示来自表23.73的相应帧缓冲相关状态。帧缓冲相关状态的值与使用该表中的简单状态查询绑定并查询帧缓冲对象时获得的值相同。这些值可以从帧缓冲对象或默认帧缓冲查询。`SAMPLES` 和 `SAMPLE_BUFFERS` 的值是根据第9.2.3.1节的描述确定的。

将帧缓冲对象的参数 `pname` 的值存储在 `params` 中。

帧缓冲对象的附件或默认帧缓冲的缓冲区可以使用以下命令查询：

```
void GetFramebufferAttachmentParameteriv(enum target, enum attachment, enum
pname, int *params);
void GetNamedFramebufferAttachmentParameteriv(uint framebuffer, enum attachment,
enum pname, int *params);
```

对于 `GetFramebufferAttachmentParameteriv`，帧缓冲对象是绑定到 `target` 的对象，`target` 必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。`FRAMEBUFFER` 等效于 `DRAW_FRAMEBUFFER`。对于 `GetNamedFramebufferAttachmentParameteriv`，`framebuffer` 可以是零，表示查询默认的绘制帧缓冲，或者是帧缓冲对象的名称。如果 `framebuffer` 为零，则查询默认的绘制帧缓冲。

如果查询默认帧缓冲，则 `attachment` 必须是表9.1中列出的值之一，根据该表中的选择，选择单个颜色、深度或模板缓冲。否则，`attachment` 必须是表9.2中列出的帧缓冲对象附件点之一。如果 `attachment` 是 `DEPTH_STENCIL_ATTACHMENT`，则必须将相同的对象绑定到帧缓冲对象的深度和模板附件点，并返回有关该对象的信息。

成功从 `Get*FramebufferAttachmentParameteriv` 返回后，如果 `pname` 是 `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`，则 `params` 将包含 `NONE`、`FRAMEBUFFER_DEFAULT`、`TEXTURE` 或 `RENDERBUFFER` 之一，用于标识包含附加图像的对象类型。`pname` 的其他接受值取决于对象的类型，如下所述。

如果 `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` 的值为 `NONE`，则以下条件之一为真：

- 没有帧缓冲对象绑定到 `target`，或者
- 绑定了默认帧缓冲，并且
    - `attachment` 是 `DEPTH` 或 `STENCIL`，并且深度或模板位的数量为零；或者

- `attachment` 不指示分配给默认帧缓冲的颜色缓冲之一。

在这种情况下，查询 `pname` 为 `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` 将返回零，所有其他查询将生成 `INVALID_OPERATION` 错误。

如果 `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` 的值不是 `NONE`，则这些查询适用于所有其他帧缓冲类型：

- 如果 `pname` 是 `FRAMEBUFFER_ATTACHMENT_RED_SIZE`、`FRAMEBUFFER_ATTACHMENT_GREEN_SIZE`、`FRAMEBUFFER_ATTACHMENT_BLUE_SIZE`、`FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE`、`FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE` 或 `FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE`，那么 `params` 将包含指定附件的相应红色、绿色、蓝色、alpha、深度或模板分量的位数。如果请求的分量在 `attachment` 中不存在，或者未为 `attachment` 指定数据存储或纹理图像，则 `params` 将包含零。
- 如果 `pname` 是 `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE`，那么 `params` 将包含指定附件的组件格式，其中浮点数为 `FLOAT`，有符号整数为 `INT`，无符号整数为 `UNSIGNED_INT`，有符号标准化固定点为 `SIGNED_NORMALIZED`，或者无符号标准化固定点为 `UNSIGNED_NORMALIZED`。如果未为 `attachment` 指定数据存储或纹理图像，则 `params` 将包含 `NONE`。由于深度+模板附件没有单一格式，因此无法执行此查询。
- 如果 `pname` 是 `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`，那么 `params` 将包含指定附件的组件编码，其中 `LINEAR` 表示线性编码，`SRGB` 表示sRGB编码的组件。仅颜色缓冲组件可以进行sRGB编码；这些组件将按照第17.3.6和17.3.7节的描述进行处理。对于默认帧缓冲，颜色编码由实现确定。对于帧缓冲对象，如果颜色附件的内部格式是第8.24节中描述的可渲染sRGB格式之一，则组件将进行sRGB编码。如果 `attachment` 不是颜色附件，或者未为 `attachment` 指定数据存储或纹理图像，则 `params` 将包含值 `LINEAR`。

9.2.3.1 多重采样查询

`SAMPLE_BUFFERS` 和 `SAMPLES` 的值控制着是否以及如何执行多重采样（详见第 14.3.1 节）。它们是由帧缓冲对象的附件或默认帧缓冲的缓冲区导出的与帧缓冲相关的常量，可以通过针对特定帧缓冲调用 `GetFramebufferParameteriv` 和 `GetNamedFramebufferParameteriv`（参见第 9.2.3 节），或者通过将 `pname` 设置为 `SAMPLE_BUFFERS` 或 `SAMPLES` 调用 `GetIntegerv` 来确定。

如果帧缓冲对象不是帧缓冲完整的（定义见第 9.4.2 节），则 `SAMPLE_BUFFERS` 和 `SAMPLES` 的值是未定义的。

否则，`SAMPLES` 的值等于 `RENDERBUFFER_SAMPLES` 或 `TEXTURE_SAMPLES`（取决于附加图像的类型），这些值必须都相同。如果 `SAMPLES` 不为零，则 `SAMPLE_BUFFERS` 的值为 1，否则为零。

9.2.4 渲染缓冲对象

渲染缓冲对象是一个包含可渲染内部格式的单个图像的数据存储对象。下面描述的命令用于分配和删除渲染缓冲对象的图像，并将渲染缓冲对象的图像附加到帧缓冲对象。

渲染缓冲对象的名称空间是无符号整数，其中零被 GL 保留。通过将由 GenRenderbuffers（见下文）返回的名称绑定到 RENDERBUFFER 来创建渲染缓冲对象。通过调用以下命令实现绑定：

```
void BindRenderbuffer( enum target, uint renderbuffer );
```

其中 `target` 设置为 `RENDERBUFFER`，`renderbuffer` 设置为渲染缓冲对象的名称。如果 `renderbuffer` 不为零，则生成的渲染缓冲对象是一个新的状态向量，初始化为零大小的内存缓冲区，包括在表 23.27 中列出的所有状态，并具有相同的初始值。任何对目标的先前绑定都会被打破。

`BindRenderbuffer` 也可以用于绑定现有的渲染缓冲对象。如果绑定成功，则不会更改新绑定的渲染缓冲对象的状态，并且对目标的任何先前绑定都将被打破。

在绑定渲染缓冲对象时，GL 对其绑定的渲染缓冲对象的目标的操作会影响到该渲染缓冲对象，并且对其绑定的目标的查询将返回来自绑定对象的状态。

名称零被保留。不能使用名称零创建渲染缓冲对象。如果 `renderbuffer` 为零，则对目标的任何先前绑定都将被打破，并且目标绑定将恢复到初始状态。

在初始状态下，保留的名称零绑定到 `RENDERBUFFER`。不存在与名称零对应的渲染缓冲对象，因此在绑定零的情况下，客户端尝试修改或查询 RENDERBUFFER 目标的渲染缓冲对象状态将生成 GL 错误，如第 9.2.3 节所述。

通过调用 `GetIntegerv` 并将 `pname` 设置为 `RENDERBUFFER_BINDING`，可以确定当前的 `RENDERBUFFER` 绑定。

新的渲染缓冲对象还可以使用以下命令创建：

```
void CreateRenderbuffers( sizei n, uint *renderbuffers );
```

`CreateRenderbuffers` 返回 `n` 个以前未使用的渲染缓冲对象名称，每个名称代表一个新的渲染缓冲对象，它是一个状态向量，包括在表 23.27 中列出的所有状态以及初始值。每个渲染缓冲对象的状态就像从 `GenRenderbuffers` 返回的名称已绑定到 `RENDERBUFFER` 目标一样，除了对 `RENDERBUFFER` 的任何现有绑定都不受影响。

命令

```
void GenRenderbuffers(sizei n, uint *renderbuffers);
```

返回n个先前未使用的渲染缓冲对象名称，这些名称仅用于`GenRenderbuffers`目的，但仅当它们首次绑定时才获取渲染缓冲对象状态。

通过调用以下命令来删除渲染缓冲对象：

```
void DeleteRenderbuffers(sizei n, const uint *renderbuffers);
```

其中`renderbuffers`包含要删除的`n`个渲染缓冲对象的名称。删除渲染缓冲对象后，它将不再包含内容，并且其名称将再次变为未使用。如果删除了当前绑定到`RENDERBUFFER`的渲染缓冲对象，则就像执行了`BindRenderbuffer`，目标为`RENDERBUFFER`，名称为零。此外，当删除渲染缓冲对象时，如果渲染缓冲对象的图像附加到帧缓冲对象（参见第9.2.7节），则必须特别小心。

未使用的名称在为`GenRenderbuffers`标记为已用时再次标记为未使用。未使用的名称在渲染缓冲对象中被静默忽略，零值也是如此。

命令

```
boolean IsRenderbuffer(uint renderbuffer);
```

如果`renderbuffer`是渲染缓冲对象的名称，则返回`TRUE`。如果`renderbuffer`为零，或者`renderbuffer`是非零且不是渲染缓冲对象的名称，则`IsRenderbuffer`返回`FALSE`。

渲染缓冲对象的图像的数据存储、格式、维度和采样数量是通过以下命令确定的：

```
void RenderbufferStorageMultisample(enum target, sizei samples, enum
internalformat, sizei width, sizei height);
```

```
void NamedRenderbufferStorageMultisample(uint renderbuffer, sizei samples, enum
internalformat, sizei width, sizei height);
```

对于`RenderbufferStorageMultisample`，渲染缓冲对象是绑定到`target`的对象，`target`必须是`RENDERBUFFER`。对于`NamedRenderbufferStorageMultisample`，`renderbuffer`是渲染缓冲对象的名称。`internalformat`必须是颜色渲染、深度渲染或模板渲染（如第9.4节中定义）。

`width`和`height`是渲染缓冲对象的图像的像素维度。

成功后，`*RenderbufferStorageMultisample`会删除渲染缓冲对象图像的任何现有数据存储，并且数据存储的内容是未定义的。`RENDERBUFFER_WIDTH`设置为`width`，`RENDERBUFFER_HEIGHT`设置为`height`，`RENDERBUFFER_INTERNAL_FORMAT`设置为`internalformat`。

如果`samples`为零，则`RENDERBUFFER_SAMPLES`设置为零。否则，`samples`表示对所需最小采样数的请求。由于不同的实现可能支持不同数量的多采样渲染，因此为渲染缓冲对象图像分配的实际采样数是依赖于实现的。但是，`RENDERBUFFER_SAMPLES`的结果值保证大于或等于`samples`，并且不超过实现支持的下一个较大的采样数。

GL实现可以根据任何`*RenderbufferStorageMultisample`参数（除了`target`和`renderbuffer`）变化其内部组件分辨率的分配，但是分配和选择的内部格式不能是任何其他状态的函数，并且一旦建立将不能更改。

命令

```
void Renderbuffer

Storage(enum target, enum internalformat, sizei width, sizei height);
```

```
void NamedRenderbufferStorage(uint renderbuffer, enum internalformat, sizei
width, sizei height);
```

相当于

```
RenderbufferStorageMultisample(target, 0, internalformat, width, height);
```

和

```
NamedRenderbufferStorageMultisample(renderbuffer, 0, internalformat, width,
height);
```

respectively

9.2.5 渲染缓冲对象所需格式

实现必须至少支持每种类型（unsigned int、float等）和每个基本内部格式的一个内部组件分辨率的分配。
此外，实现必须支持以下大小和压缩的内部格式。为渲染缓冲对象请求这些大小的内部格式将至少分配内部组件的大小，并且完全与相应表中所示格式的组件类型相匹配：

- 在表8.12的“Req. rend.”列中进行了检查的颜色格式。
- 在表8.13的“Req. format”列中进行了检查的深度、深度+模板和模板格式。
  渲染缓冲对象所需的颜色格式是纹理所需格式的子集（参见第8.5.1节）。
  实现必须支持使用最多MAX_SAMPLES个多重采样创建这些所需格式的渲染缓冲对象，但带符号和无符号整数格式只需要支持最多MAX_INTEGER_SAMPLES个多重采样的渲染缓冲对象的创建，其值必须至少为一。

9.2.6 渲染缓冲对象查询

可以使用以下命令查询渲染缓冲对象的参数：

```
void GetRenderbufferParameteriv( enum target, enum pname, int *params );
void GetNamedRenderbufferParameteriv( uint renderbuffer, enum pname, int *params
);
```

对于 `GetRenderbufferParameteriv`，渲染缓冲对象是绑定到 `target` 的对象，`target` 必须是 `RENDERBUFFER`。对于 `GetNamedRenderbufferParameteriv`，`renderbuffer` 是渲染缓冲对象的名称。

渲染缓冲对象的参数 `pname` 的值将存储在 `params` 中。`pname` 必须是表 23.27 中的符号值之一。

- 如果 `pname` 是 `RENDERBUFFER_WIDTH`、`RENDERBUFFER_HEIGHT`、`RENDERBUFFER_INTERNAL_FORMAT` 或 `RENDERBUFFER_SAMPLES`，则 `params` 将包含渲染缓冲对象图像的宽度（以像素为单位）、高度（以像素为单位）、内部格式或采样数量。
- 如果 `pname` 是 `RENDERBUFFER_RED_SIZE`、`RENDERBUFFER_GREEN_SIZE`、`RENDERBUFFER_BLUE_SIZE`、`RENDERBUFFER_ALPHA_SIZE`、`RENDERBUFFER_DEPTH_SIZE` 或 `RENDERBUFFER_STENCIL_SIZE`，则 `params` 将包含渲染缓冲对象图像的红色、绿色、蓝色、Alpha、深度或模板组件的实际分辨率（而不是在定义图像时指定的分辨率）。

9.2.7 将渲染缓冲图像附加到帧缓冲

可以使用以下命令将渲染缓冲对象附加为帧缓冲对象的逻辑缓冲之一：

```
void FramebufferRenderbuffer( enum target, enum attachment, enum
renderbuffertarget, uint renderbuffer );
void NamedFramebufferRenderbuffer( uint framebuffer, enum attachment, enum
renderbuffertarget, uint renderbuffer );
```

对于 `FramebufferRenderbuffer`，帧缓冲对象是绑定到 `target` 的对象，`target` 必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。`FRAMEBUFFER` 等同于 `DRAW_FRAMEBUFFER`。
对于 `NamedFramebufferRenderbuffer`，`framebuffer` 是帧缓冲对象的名称。`attachment` 必须设置为表9.2中列出的帧缓冲的附加点之一。
`renderbuffertarget` 必须是 `RENDERBUFFER`，而 `renderbuffer` 可以是零，也可以是要附加到帧缓冲的类型为 `renderbuffertarget` 的渲染缓冲对象的名称。如果 `renderbuffer` 是零，则忽略 `renderbuffertarget` 的值。
如果 `renderbuffer` 不是零且 `FramebufferRenderbuffer` 成功，那么名为 `renderbuffer` 的渲染缓冲将用作帧缓冲对象附加点标识的逻辑缓冲。指定附加点的 `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` 的值设置为 `RENDERBUFFER`，`FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` 的值设置为 `renderbuffer`。指定的附加点的所有其他状态值都设置为它们在表23.25中列出的默认值。不更改渲染缓冲对象的状态，且将断开对帧缓冲对象的附加到该附加逻辑缓冲的任何先前附加。如果附加不成功，则不更改渲染缓冲

对象或帧缓冲对象的状态。
使用渲染缓冲名称为零调用 `FramebufferRenderbuffer` 将分离帧缓冲对象当前绑定到 `target` 的附加点标识的任何图像，如果存在的话。帧缓冲对象中指定的附加点的所有状态值都设置为它们在表23.25中列出的默认值。
将 `attachment` 设置为值 `DEPTH_STENCIL_ATTACHMENT` 是一种特殊情况，导致帧缓冲对象的深度和模板附加点都设置为 `renderbuffer`，该 `renderbuffer` 应该具有基本内部格式 `DEPTH_STENCIL`。
如果删除渲染缓冲对象时其图像附加到当前绑定帧缓冲对象中的一个或多个附加点，则就好像对该图像在该帧缓冲对象中附加的每个附加点调用了 `FramebufferRenderbuffer`，带有零的渲染缓冲对象。换句话说，首先从该帧缓冲对象中的所有附加点分离渲染缓冲图像。请注意，渲染缓冲图像明确不会从任何未绑定的帧缓冲对象中分离。从任何未绑定的帧缓冲对象中分离图像是应用程序的责任。

9.2.8 将纹理图像附加到帧缓冲

OpenGL支持通过使用 `CopyTexImage*` 和 `CopyTexSubImage*` 等例程将帧缓冲的渲染内容复制到纹理对象的图像中。此外，OpenGL还支持直接渲染到纹理对象的图像。

要直接渲染到纹理图像，可以使用以下命令之一将纹理对象的指定级别附加为帧缓冲对象的逻辑缓冲之一：

```
void FramebufferTexture( enum target, enum attachment, uint texture, int level );
void NamedFramebufferTexture( uint framebuffer, enum attachment, uint texture,
int level );
```

对于 `FramebufferTexture`，帧缓冲对象是绑定到 `target` 的对象，`target` 必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。`FRAMEBUFFER` 等效于 `DRAW_FRAMEBUFFER`。对于 `NamedFramebufferTexture`，`framebuffer` 是帧缓冲对象的名称。`attachment` 必须是表9.2中列出的帧缓冲的附加点之一。

如果 `texture` 非零，则将纹理对象 `texture` 的指定 mip 映射级别附加到由 `attachment` 指定的帧缓冲附加点。如果 `texture` 是三维纹理、立方体贴图数组纹理、立方体贴图、一维或二维数组纹理或二维多重采样数组纹理的名称，则附加到帧缓冲附加点的纹理级别是图像的数组，且帧缓冲附加被视为分层的。

此外，可以使用以下命令之一将纹理对象的指定图像附加为帧缓冲对象的逻辑缓冲之一：

```
void FramebufferTexture1D( enum target, enum attachment, enum textarget, uint
texture, int level );
void FramebufferTexture2D( enum target, enum attachment, enum textarget, uint
texture, int level );
void FramebufferTexture3D( enum target, enum attachment, enum textarget, uint
texture, int level, int layer );
```

`target` 指定帧缓冲对象绑定的目标，必须是 `DRAW_FRAMEBUFFER`、`READ_FRAMEBUFFER` 或 `FRAMEBUFFER`。`FRAMEBUFFER` 等效于 `DRAW_FRAMEBUFFER`。

`attachment` 必须是表9.2中列出的帧缓冲的附加点之一。

如果 `texture` 不为零，则 `texture` 必须是具有 `textarget` 目标的现有纹理对象的名称，或者 `texture` 必须是一个现有立方体贴图纹理的名称，且 `textarget` 必须是表8.19中的一个立方体贴图面目标。

`level` 指定要附加到帧缓冲的纹理图像的 mipmap 级别，必须满足以下条件：

- 如果 `texture` 引用的是一个不可变格式的纹理，`level` 必须大于等于零且小于 `texture` 的 `TEXTURE_IMMUTABLE_LEVELS` 值。
- 如果 `textarget` 是 `TEXTURE_RECTANGLE` 或 `TEXTURE_2D_MULTISAMPLE`，则 `level` 必须为零。
- 如果 `textarget` 是 `TEXTURE_3D`，则 `level` 必须大于等于零且小于等于 `MAX_3D_TEXTURE_SIZE` 的 `log2` 值。
- 如果 `textarget` 是表8.19中的一个立方体贴图面目标，那么 `level` 必须大于等于零且小于等于 `MAX_CUBE_MAP_TEXTURE_SIZE` 的 `log2` 值。
- 对于 `textarget` 的所有其他值，`level` 必须大于等于零且不大于 `MAX_TEXTURE_SIZE` 的 `log2` 值。

`layer` 指定三维纹理中二维图像的层。

可以使用以下命令之一将三维或数组纹理对象的单个层附加为帧缓冲对象的逻辑缓冲之一：

```
void FramebufferTextureLayer( enum target, enum attachment, uint texture, int
level, int layer );
void NamedFramebufferTextureLayer( uint framebuffer, enum attachment, uint
texture, int level, int layer );
```

这些命令与等效的 `FramebufferTexture` 和 `NamedFramebufferTexture` 命令完全相同，除了额外的 `layer` 参数，它选择要附加的纹理对象的层。

`layer` 指定了纹理中一维或二维图像的层，但不包括立方体贴图和立方体贴图数组纹理。对于立方体贴图纹理，`layer` 被转换为如表9.3中所述的立方体贴图面。对于立方体贴图数组纹理，`layer` 被转换为一个数组层和一个立方体贴图面，如第8.5.3节中关于层-面编号的描述。

`level` 指定要附加到帧缓冲的纹理图像的mipmap级别，并且必须满足以下条件：

- 如果纹理引用的是一个不可变格式的纹理，`level` 必须大于等于零且小于 TEXTURE_IMMUTABLE_LEVELS 的值。
- 如果纹理是一个三维纹理，那么 `level` 必须大于等于零且小于等于 log2(MAX_3D_TEXTURE_SIZE)。
- 如果纹理是一个二维数组纹理，那么 `level` 必须大于等于零且小于等于 log2(MAX_TEXTURE_SIZE)。
- 如果纹理是一个二维多重采样数组纹理，那么 `level` 必须为零。

如果纹理非零且命令没有导致错误，与附件对应的帧缓冲附件状态将如其他 FramebufferTexture* 命令一样更新，除了 FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER 的值被设置为 `layer`。

9.2.8.1 附加纹理图像的效果

本节中剩余的评论适用于所有形式的 *FramebufferTexture*。

如果纹理为零，则从由 `attachment` 指定的附加点分离任何附加到该点的图像或图像数组。当纹理为零时，忽略任何附加的额外参数（`level`、`textarget` 和/或 `layer`）。 `attachment` 指定的附加点的所有状态值都设置为表 23.25 中列出的其默认值。

如果纹理不为零，并且 *FramebufferTexture* 成功，那么指定的纹理图像将用作当前绑定到 `target` 的帧缓冲对象的由 `attachment` 标识的逻辑缓冲。指定附加点的状态值设置如下：

- FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE 的值设置为 TEXTURE。
- FRAMEBUFFER_ATTACHMENT_OBJECT_NAME 的值设置为 `texture`。

- FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL 的值设置为 `level`。
- 如果调用 *FramebufferTexture2D 且纹理为立方体贴图，则 FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE 的值设置为 `textarget`；否则设置为默认值（NONE）。
- 如果调用 *FramebufferTextureLayer 或 *FramebufferTexture3D，则 FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER 的值设置为 `layer`；否则设置为零。
- 如果调用 *FramebufferTexture* 且纹理为三维、立方体贴图、二维多重采样数组或一维或二维数组纹理的名称，则 FRAMEBUFFER_ATTACHMENT_LAYERED 的值设置为 TRUE；否则设置为 FALSE。

附加点 `attachment` 指定的附加点的所有其他状态值都设置为表 23.25 中列出的其默认值。不会更改纹理对象的状态，并且任何之前对绑定到 `framebuffer target` 的帧缓冲对象的附加都会被断开。如果附加不成功，则纹理对象或帧缓冲对象的状态都不会更改。

将 `attachment` 设置为 DEPTH_STENCIL_ATTACHMENT 是一种特殊情况，它导致帧缓冲对象的深度和模板附加点都设置为 `texture`。 `texture` 必须具有基本内部格式 DEPTH_STENCIL，否则深度和模板帧缓冲附加将不完整（参见第 9.4.1 节）。

如果在其图像附加到当前绑定的帧缓冲对象中的一个或多个附加点的情况下删除纹理对象，则就像对该帧缓冲对象的每个附加点都调用了 *FramebufferTexture* 一样。换句话说，首先将纹理图像从该帧缓冲对象的所有附加点分离。请注意，纹理图像特别不会从任何未绑定的帧缓冲对象分离。将纹理图像从任何未绑定的帧缓冲对象分离是应用程序的责任。

9.3 纹理和帧缓冲之间的反馈循环

当纹理对象同时用作 GL 操作的源和目标时，可能存在反馈循环。存在反馈循环时，将导致未定义的行为。本节更详细地描述了渲染反馈循环（参见第 8.14.2.1 节）和纹理复制反馈循环（参见第 8.6.1 节）。

###9.3.1 渲染反馈循环

将一维或二维纹理级别、立方体贴图纹理级别的一个面或二维数组或三维纹理的一个层附加到绘制帧缓冲时，不会阻止相同纹理同时绑定到纹理单元。在这种情况下，访问该图像的纹理操作将产生未定义的结果，如第 8.14 节末尾所述。导致此类未定义行为的条件在下面更详细地定义。应该避免这种未定义的纹理操作，这些操作可能使片段处理操作的最终结果变得未定义。

在绑定纹理对象的同时，必须采取特殊预防措施，以避免将纹理图像附加到当前绑定的绘制帧缓冲对象。这样做可能导致 GL 渲染操作写入像素，并在将其用作当前绑定纹理中的纹素时同时读取这些相同的像素。在这种情况下，帧缓冲将被视为帧缓冲完整（参见第 9.4 节），但在处于此状态时渲染的片段的值将是未定义的。纹理样本的值也可能是未定义的，如第 8.14.2.1 节“渲染反馈循环”下所述。

具体来说，如果任何着色器阶段提取纹素并通过片段着色器输出写入相同的纹素，则渲染的片段的值是未定义的，即使读取和写入不在同一绘制调用中，除非以下任何异常情况适用：

- 读取和写入是来自/写入不相交纹素集的（考虑到纹理过滤规则）。
- 只有每个纹素的单一读取和写入，而读取是写入相同纹素的片段着色器调用（例如使用 `texelFetch2D(sampler, ivec2(gl_FragCoord.xy), 0);`）。
- 如果写入了一个纹素，则为了安全地读取结果，纹素获取必须在由命令 `void TextureBarrier( void );` 分隔的后续绘制调用中。 `TextureBarrier` 将确保写入已完成，并在执行后续绘制调用之前使缓存无效。

9.3.2 纹理复制反馈循环

与渲染反馈循环类似，当将纹理图像附加到当前绑定的读取帧缓冲对象时，同时该纹理图像又是`CopyTexImage*`操作的目标时，可能会存在纹理复制反馈循环，详见第8.6.1节“纹理复制反馈循环”。在这种情况下，由复制操作写入的纹素值可能是未定义的（与通过`BlitFramebuffer`进行的重叠复制类似）。

具体来说，如果以下所有条件都为真，则所复制纹素的值是未定义的：

- 来自纹理对象T的图像已附加到当前绑定的读取帧缓冲对象的附件点A。
- 所选的读取缓冲（参见第18.2.1节）是附件点A。
- T已绑定到`CopyTexImage*`操作的纹理目标。
- 复制操作的级别参数选择了与A附加的相同图像。

9.4 帧缓冲完整性

为了有效地用作GL的绘制或读取帧缓冲，帧缓冲必须是帧缓冲完整的。

默认帧缓冲如果存在则始终是完整的；然而，如果没有默认帧缓冲存在（GL上下文未关联窗口系统提供的可绘制对象），则被视为不完整。

帧缓冲对象被称为帧缓冲完整，如果其所有附加的图像和用于渲染和读取的所有帧缓冲参数都是一致定义的，并且满足以下定义的要求。帧缓冲完整性的规则取决于附加图像的属性以及某些依赖于实现的限制。

附加图像的内部格式可以影响帧缓冲的完整性，因此首先定义图像的内部格式与其可以附加到的附件点之间的关系。

- 如果内部格式是RED、RG、RGB、RGBA，或表8.12中“CR”（颜色渲染）列中已选中的表8.12的大小内部格式之一，则其为颜色可渲染格式。没有其他格式，包括压缩的内部格式，是颜色可渲染的。
- 如果内部格式为DEPTH_COMPONENT或表8.13中其基本内部格式为DEPTH_COMPONENT或DEPTH_STENCIL的格式之一，则其为深度可渲染格式。没有其他格式是深度可渲染的。
- 如果内部格式为STENCIL_INDEX、DEPTH_STENCIL，或表8.13中其基本内部格式为STENCIL_INDEX或DEPTH_STENCIL的格式之一，则其为模板可渲染格式。没有其他格式是模板可渲染的。

9.5 像素与附加图像中元素的映射

当DRAW_FRAMEBUFFER_BINDING非零时，写入帧缓冲的操作会修改附加到所选逻辑缓冲的图像，而从帧缓冲中读取的操作则从附加到所选逻辑缓冲的图像中读取。

如果附加的图像是渲染缓冲图像，则窗口坐标（xw，yw）对应于渲染缓冲图像中相同坐标的值。

如果附加的图像是纹理图像，则窗口坐标（xw，yw）与图8.3中的纹理单元（i，j，k）对应如下：

- i = xw
- j = yw
- k = layer

其中，layer是所选逻辑缓冲的FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER的值。对于二维纹理，k和layer无关紧要；对于一维纹理，j，k和layer无关紧要。

如果 xw、yw 或 layer 小于零，或者 xw、yw 或 layer 分别大于或等于纹理图像的宽度、高度或深度，则 (xw, yw) 对应于边界纹素。

9.6 转换为帧缓冲可附加图像组件

当绘制帧缓冲绑定为非零时，对于每个绘制缓冲，启用的颜色值被写入帧缓冲。R、G、B 和 A 值根据表 8.11 描述的内部格式转换为内部组件，具体取决于所选逻辑缓冲附加图像的内部格式。然后，将得到的内部组件写入逻辑缓冲附加的图像。在此过程中，17.4.2 节中描述的掩码操作也是有效的。

9.7 转换为 RGBA 值

当读取帧缓冲绑定为非零时，或者在绘制帧缓冲绑定为非零时作为逻辑操作的源或混合的源时，从所选逻辑缓冲附加的图像获取的颜色值的组件首先根据表 15.1和附加图像的内部格式转换为 R、G、B 和 A 值。

9.8 分层帧缓冲

当一个帧缓冲是完整的并且所有已填充的附件都是分层的时，它被认为是分层的。在向分层帧缓冲中渲染时，GL 生成的每个片段都被分配一个层编号。如果满足以下条件之一，则片段的层编号为零：

- 几何着色器被禁用，或者
- 当前几何着色器未静态分配值给内建输出变量 gl_Layer。

否则，由几何着色器发射的每个点、线或三角形的层都取自于原始几何体的一个顶点的 gl_Layer 输出。所使用的顶点取决于实现。为了得到定义良好的结果，每个发射的原始几何体的所有顶点都应该为 gl_Layer 设置相同的值。由于 EndPrimitive 内建函数开始一个新的输出原始体，可以在两个发射不同层编号的顶点之间调用 EndPrimitive 来获得定义良好的结果。如果帧缓冲不是分层的，则几何着色器写入的层编号不起作用。

当片段写入分层帧缓冲时，片段的层编号选择用于模板测试（参见17.3.3 节）、深度缓冲测试（参见17.3.4 节）以及混合和颜色缓冲写入（参见17.3.6 节）的附件数组中的图像。如果片段的层编号为负数，或者大于或等于任何附件的最小层数，则片段对帧缓冲内容的影响是未定义的。

当使用17.4.3 节中描述的 Clear 或 ClearBuffer* 命令来清除分层帧缓冲附件时，会清除附件的所有层。

当像 ReadPixels 等命令从分层帧缓冲中读取时，所选附件的零层图像总是用于获取像素值。

当立方图纹理级别被附加到分层帧缓冲时，有六个层，编号从零到五。每个层编号对应一个立方体贴图面，如表 9.3 所示。

当立方图数组纹理级别被附加到分层帧缓冲时，层编号对应于一个层-面。层-面可以通过以下方式转换为数组层和立方图面：

- 数组层 = floor(layer / 6)
- 面 = layer mod 6

其中，面编号对应于表 9.3 中显示的立方图面。

mesa-18 源码分析

pipe 层接口分析

pipe 层fb定义

state_tracker定义同

```
/**
 * 帧缓冲是渲染缓冲的集合（颜色、深度、模板等）。
 * 从C++的角度来看，可以将其视为设备驱动程序将生成的派生类的基类。
 */
struct gl_framebuffer
{
   /**
    * 如果为零，则这是一个窗口系统帧缓冲。如果非零，则这是一个FBO帧缓冲；
    * 注意对于一些设备（即那些对于FBO具有与OpenGL/Mesa坐标系统不同的自然像素坐标系统的设备），
    * 这意味着视口、多边形面方向和多边形点画需要进行反转。
    */
   GLuint Name;
   GLint RefCount;


   GLboolean DeletePending;

   /**
    * 帧缓冲的视觉。如果这是窗口系统缓冲，它是不可变的。如果是用户创建的FBO，则从附件计算。
    */
   struct gl_config Visual;

   /**
    * 帧缓冲的大小（以像素为单位）。如果没有附件，这两者都为0。
    */
   GLuint Width, Height;

   /**
    * 如果帧缓冲没有附件（即GL_ARB_framebuffer_no_attachments），则帧缓冲的几何形状由默认
值指定。
    */
   struct {
     GLuint Width, Height, Layers, NumSamples;
     GLboolean FixedSampleLocations;
     /* 由驱动程序从NumSamples派生，以便它可以选择硬件的有效值。 */
     GLuint _NumSamples;
   } DefaultGeometry;

   /** \name  绘制边界（缓冲大小和剪切框的交集）
    * 绘制区域由[_Xmin，_Xmax) x [_Ymin，_Ymax)给出，
    *  （对于_Xmin和_Ymin是包含的，而对于_Xmax和_Ymax是不包含的）
    */
   /*@{*/
   GLint _Xmin, _Xmax;
   GLint _Ymin, _Ymax;
```

```
   /*@}*/

   /** \name  派生的Z缓冲信息 */
   /*@{*/
   GLuint _DepthMax;    /**< 最大深度缓冲值 */
   GLfloat _DepthMaxF;  /**< 浮点型最大深度缓冲值 */
   GLfloat _MRD;    /**< Z值的最小可分辨差异 */
   /*@}*/

   /** GL_FRAMEBUFFER_(IN)COMPLETE_* tokens之一 */
   GLenum16 _Status;

   /** 是否存在Attachment的Type != GL_NONE
    * 注意：在没有附件的情况下，_HasAttachments标志的值为false，
    * 支持GL_ARB_framebuffer_no_attachments扩展的后端驱动程序必须检查_HasAttachments标
志，
    * 如果为GL_FALSE，则必须使用DefaultGeometry中的值来初始化其视口、剪切等（特别是_Xmin、
_Xmax、_Ymin和_Ymax不考虑_HasAttachments为false的情况）。
    * 要获取帧缓冲的几何信息，可用的辅助函数有
    *   _mesa_geometric_width(),
    *   _mesa_geometric_height(),
    *   _mesa_geometric_samples()和
    *   _mesa_geometric_layers(),
    * 它们检查_HasAttachments。
    */
   bool _HasAttachments;

   GLbitfield _IntegerBuffers;  /**< 哪些颜色缓冲区是整数值 */

   /* ARB_color_buffer_float */
   GLboolean _AllColorBuffersFixedPoint; /* 既不是整数也不是浮点数 */
   GLboolean _HasSNormOrFloatColorBuffer;

   /**
    * 帧缓冲中的最大层数，如果帧缓冲不是分层的，则为0。
    * 对于立方体贴图和立方体贴图数组，每个立方体面都算作一层。
    * 与Width、Height一样，支持GL_ARB_framebuffer_no_attachments的后端驱动程序必须在
_HasAttachments为false的情况下使用DefaultGeometry。
    */
   GLuint MaxNumLayers;

   /** 所有渲染缓冲附件的数组，由BUFFER_*标记索引。*/
   struct gl_renderbuffer_attachment Attachment[BUFFER_COUNT];

   /* 在未扩展的OpenGL中，这些变量是GL_COLOR_BUFFER属性组和GL_PIXEL属性组的一部分。 */
   GLenum16 ColorDrawBuffer[MAX_DRAW_BUFFERS];
   GLenum16 ColorReadBuffer;

   /* GL_ARB_sample_locations */
   GLfloat *SampleLocationTable; /**< 如果为NULL，则没有指定表 */
   GLboolean ProgrammableSampleLocations;
   GLboolean SampleLocationPixelGrid;

   /** 从上述ColorDraw/ReadBuffer计算而来 */
   GLuint _NumColorDrawBuffers;
   gl_buffer_index _ColorDrawBufferIndexes[MAX_DRAW_BUFFERS];
```

```
   gl_buffer_index _ColorReadBufferIndex;
   struct gl_renderbuffer *_ColorDrawBuffers[MAX_DRAW_BUFFERS];
   struct gl_renderbuffer *_ColorReadBuffer;

   /* GL_MESA_framebuffer_flip_y */
   bool FlipY;

   /** 删除此帧缓冲 */
   void (*Delete)(struct gl_framebuffer *fb);
};
```

创建fbo(非dsa机制)

```
_mesa_GenFramebuffers(GLsizei n, GLuint *framebuffers)
{
    //This is the implementation for glGenFramebuffers and glCreateFramebuffers.
   create_framebuffers(n, framebuffers, false);
}

static void
create_framebuffers(GLsizei n, GLuint *framebuffers, bool dsa)
{
   struct gl_framebuffer *fb;
   first = _mesa_HashFindFreeKeyBlock(ctx->Shared->FrameBuffers, n);
   for (i = 0; i < n; i++)
      GLuint name = first + i;
      framebuffers[i] = name;
      if (dsa) {
         fb = ctx->Driver.NewFramebuffer(ctx, framebuffers[i]);
            [jump state_tracker _mesa_new_framebuffer ]
      }
      else
         fb = &DummyFramebuffer;
      _mesa_HashInsertLocked(ctx->Shared->FrameBuffers, name, fb);
}

void GLAPIENTRY
_mesa_BindFramebuffer(GLenum target, GLuint framebuffer)
{
   /* OpenGL ES glBindFramebuffer and glBindFramebufferOES use this same entry
    * point, but they allow the use of user-generated names.
    */
   bind_framebuffer(target, framebuffer);
}

static void
bind_framebuffer(GLenum target, GLuint framebuffer)
{
   struct gl_framebuffer *newDrawFb, *newReadFb;
   GLboolean bindReadBuf, bindDrawBuf;
   GET_CURRENT_CONTEXT(ctx);

   switch (target) {
   case GL_DRAW_FRAMEBUFFER_EXT:
```

```
      bindDrawBuf = GL_TRUE;
      bindReadBuf = GL_FALSE;
      break;
   case GL_READ_FRAMEBUFFER_EXT:
      bindDrawBuf = GL_FALSE;
      bindReadBuf = GL_TRUE;
      break;
   case GL_FRAMEBUFFER_EXT:
      bindDrawBuf = GL_TRUE;
      bindReadBuf = GL_TRUE;
      break;
   default:
      _mesa_error(ctx, GL_INVALID_ENUM, "glBindFramebufferEXT(target)");
      return;
   }

   if (framebuffer) {
      /* Binding a user-created framebuffer object */
      newDrawFb = _mesa_lookup_framebuffer(ctx, framebuffer);
      if (newDrawFb == &DummyFramebuffer) {
         /* ID was reserved, but no real framebuffer object made yet */
         newDrawFb = NULL;
      }
      if (!newDrawFb) {
         /* create new framebuffer object */
         newDrawFb = ctx->Driver.NewFramebuffer(ctx, framebuffer);
         if (!newDrawFb) {
            _mesa_error(ctx, GL_OUT_OF_MEMORY, "glBindFramebufferEXT");
            return;
         }
         _mesa_HashInsert(ctx->Shared->FrameBuffers, framebuffer, newDrawFb);
      }
      newReadFb = newDrawFb;
   }
   else {
      /* Binding the window system framebuffer (which was originally set
       * with MakeCurrent).
       */
      newDrawFb = ctx->WinSysDrawBuffer;
      newReadFb = ctx->WinSysReadBuffer;
   }

   _mesa_bind_framebuffers(ctx,
                           bindDrawBuf ? newDrawFb : ctx->DrawBuffer,
                           bindReadBuf ? newReadFb : ctx->ReadBuffer);

    struct gl_framebuffer *const oldDrawFb = ctx->DrawBuffer;
    struct gl_framebuffer *const oldReadFb = ctx->ReadBuffer;
    const bool bindDrawBuf = oldDrawFb != newDrawFb;
    const bool bindReadBuf = oldReadFb != newReadFb;
    if (bindReadBuf) {
       FLUSH_VERTICES(ctx, _NEW_BUFFERS);
       _mesa_reference_framebuffer(&ctx->ReadBuffer, newReadFb);
    }
    if (bindDrawBuf) {
       FLUSH_VERTICES(ctx, _NEW_BUFFERS);
```

```
          ctx->NewDriverState |= ctx->DriverFlags.NewSampleLocations;

          /* check if newly bound framebuffer has any texture attachments */
          check_begin_texture_render(ctx, newDrawFb);
               assert(ctx->Driver.RenderTexture);
               if (_mesa_is_winsys_fbo(fb))
                  return; /* can't render to texture with winsys framebuffers */

               for (i = 0; i < BUFFER_COUNT=16; i++) {
                  struct gl_renderbuffer_attachment *att = fb->Attachment + i;
                  if (att->Texture && att->Renderbuffer->TexImage
                      && driver_RenderTexture_is_safe(att)) {
                     ctx->Driver.RenderTexture(ctx, fb, att);
                  }
               }

          _mesa_reference_framebuffer(&ctx->DrawBuffer, newDrawFb);
       }
}
```

- 绑定READ_FRAMEBUFFER设置状态_NEW_BUFFERS, 将newDrawFb 绑定到gl_context.ReadBuffer,
- 绑定DRAW_FRAMEBUFFER设置状态_NEW_BUFFERS,设置驱动状态ST_NEW_SAMPLE_STATE, 将newDrawFb 绑定到gl_context.DrawBuffer,
- winsy fb不能渲染到纹理

附件gl_renderbuffer_attachment定义

```
/**
 * 渲染缓冲附件指向纹理对象（并指定了一个mipmap级别、立方体面或3D纹理切片），
 * 或指向渲染缓冲。
 */
struct gl_renderbuffer_attachment
{
   GLenum16 Type; /**< \c GL_NONE或\c GL_TEXTURE或\c GL_RENDERBUFFER_EXT */
   GLboolean Complete;

   /**
    * 如果\c Type是\c GL_RENDERBUFFER_EXT，这存储指向应用提供的渲染缓冲对象的指针。
    */
   struct gl_renderbuffer *Renderbuffer;

   /**
    * 如果\c Type是\c GL_TEXTURE，这存储指向应用提供的纹理对象的指针。
    */
   struct gl_texture_object *Texture;
   GLuint TextureLevel; /**< 附加的mipmap级别。 */
   GLuint CubeMapFace;  /**< 立方体贴图的0 .. 5。 */
   GLuint Zoffset;      /**< 3D纹理的切片，或1D和2D数组纹理的层 */
   GLboolean Layered;
};
```

pipe层 rbo gl_renderbuffer定义

```
/**
 * Renderbuffers表示绘图表面，例如颜色、深度和/或模板。帧缓冲对象包含一组渲染缓冲。
 * 驱动程序通常会派生出这种类型的子类。
 */
struct gl_renderbuffer
{
   simple_mtx_t Mutex; /**< 用于线程安全的互斥锁 */
   GLuint ClassID;        /**< 驱动程序有用的类标识符 */
   GLuint Name;
   GLchar *Label;         /**< GL_KHR_debug */
   GLint RefCount;
   GLuint Width, Height;
   GLuint Depth;
   GLboolean Purgeable;  /**< 内存压力下是否可清除的标志 */
   GLboolean AttachedAnytime; /**< 如果它曾附加到帧缓冲，则为TRUE */

   /**
    * 对于包装纹理的渲染缓冲，为驱动程序提供刷新渲染缓存的机会。
    *
    * 驱动程序还可以在由glFramebufferTexture()生成的以外的渲染缓冲上设置此标志，
    * 尽管这意味着将在没有rb->TexImage的情况下调用FinishRenderTexture()。
    */
   GLboolean NeedsFinishRenderTexture;

   GLubyte NumSamples;    /**< 零表示不是多重采样 */
   GLubyte NumStorageSamples; /**< 用于AMD_framebuffer_multisample_advanced扩展 */
   GLenum16 InternalFormat; /**< 用户指定的格式 */
   GLenum16 _BaseFormat;    /**< 可能是GL_RGB、GL_RGBA、GL_DEPTH_COMPONENT或
GL_STENCIL_INDEX。 */
   mesa_format Format;      /**< 实际的渲染缓冲内存格式 */

   /**
    * 如果此渲染缓冲包装纹理，则指向纹理图像的指针，否则为NULL。
    *
    * 请注意，持有包含此TexImage的gl_texture_object的引用的是
gl_renderbuffer_attachment。
    */
   struct gl_texture_image *TexImage;

   /** 删除此渲染缓冲 */
   void (*Delete)(struct gl_context *ctx, struct gl_renderbuffer *rb);

   /** 为此渲染缓冲分配新的存储空间 */
   GLboolean (*AllocStorage)(struct gl_context *ctx,
                             struct gl_renderbuffer *rb,
                             GLenum internalFormat,
                             GLuint width, GLuint height);
};
```

创建Renderbuffer(dsa机制)

```
void GLAPIENTRY
_mesa_CreateRenderbuffers_no_error(GLsizei n, GLuint *renderbuffers)
{
   GET_CURRENT_CONTEXT(ctx);
   create_render_buffers(ctx, n, renderbuffers, true);
        const char *func = dsa ? "glCreateRenderbuffers" : "glGenRenderbuffers";

        for (i = 0; i < n; i++) {
           GLuint name = first + i;
           renderbuffers[i] = name;
           allocate_renderbuffer_locked(ctx, name, func);
                struct gl_renderbuffer *newRb;
                /* create new renderbuffer object */
                newRb = ctx->Driver.NewRenderbuffer(ctx, renderbuffer);
                    [jump state_tracker st_new_renderbuffer]
        }


        allocate_renderbuffer_locked(ctx, name, func);
}
```

渲染缓冲参数指定,同时分配内部存储

```
_mesa_NamedRenderbufferStorage(GLuint renderbuffer, GLenum internalformat,
                               GLsizei width, GLsizei height)
{
   /* GL_ARB_fbo says calling this function is equivalent to calling
    * glRenderbufferStorageMultisample() with samples=0.  We pass in
    * a token value here just for error reporting purposes.
    */
   renderbuffer_storage_named(renderbuffer, internalformat, width, height,
                              NO_SAMPLES, 0, "glNamedRenderbufferStorage");

        renderbuffer_storage(ctx, rb, internalFormat, width, height, samples,
                             storageSamples, func);

            _mesa_renderbuffer_storage(ctx, rb, internalFormat, width, height,
samples,
                                       storageSamples);
}




void
_mesa_renderbuffer_storage(struct gl_context *ctx, struct gl_renderbuffer *rb,
                           GLenum internalFormat, GLsizei width,
                           GLsizei height, GLsizei samples,
                           GLsizei storageSamples)
{
   const GLenum baseFormat = _mesa_base_fbo_format(ctx, internalFormat);
```

```
   FLUSH_VERTICES(ctx, _NEW_BUFFERS);

   if (rb->InternalFormat == internalFormat &&
       rb->Width == (GLuint) width &&
       rb->Height == (GLuint) height &&
       rb->NumSamples == samples &&
       rb->NumStorageSamples == storageSamples) {
      /* no change in allocation needed */
      return;
   }

   /* These MUST get set by the AllocStorage func */
   rb->Format = MESA_FORMAT_NONE;
   rb->NumSamples = samples;
   rb->NumStorageSamples = storageSamples;

   /* Now allocate the storage */
   assert(rb->AllocStorage);
   if (rb->AllocStorage(ctx, rb, internalFormat, width, height)) {
        [jump state tracker st_renderbuffer_alloc_storage]
      /* No error - check/set fields now */
      /* If rb->Format == MESA_FORMAT_NONE, the format is unsupported. */
      assert(rb->Width == (GLuint) width);
      assert(rb->Height == (GLuint) height);
      rb->InternalFormat = internalFormat;
      rb->_BaseFormat = baseFormat;
      assert(rb->_BaseFormat != 0);
   }
   else {
      /* Probably ran out of memory - clear the fields */
      rb->Width = 0;
      rb->Height = 0;
      rb->Format = MESA_FORMAT_NONE;
      rb->InternalFormat = GL_NONE;
      rb->_BaseFormat = GL_NONE;
      rb->NumSamples = 0;
      rb->NumStorageSamples = 0;
   }

   /* Invalidate the framebuffers the renderbuffer is attached in. */
   if (rb->AttachedAnytime) {
      _mesa_HashWalk(ctx->Shared->FrameBuffers, invalidate_rb, rb);
   }
}
```

将纹理附加到帧缓冲 FramebufferTexture*

FramebufferTexture

```
void GLAPIENTRY
_mesa_FramebufferTexture_no_error(GLenum target, GLenum attachment,
                                  GLuint texture, GLint level)
{
   frame_buffer_texture(0, target, attachment, texture, level, 0,
```

```
                        "glFramebufferTexture", false, true, true);

      /* Get the framebuffer object */
      struct gl_framebuffer *fb = lookup ...

      /* Get the texture object and framebuffer attachment*/
      struct gl_renderbuffer_attachment *att;
      struct gl_texture_object *texObj;
      texObj = get_texture_for_framebuffer(ctx, texture);
           return _mesa_lookup_texture(ctx, texture);
      att = get_attachment(ctx, fb, attachment, NULL);
           switch (attachment)
           case GL_COLOR_ATTACHMENT0_EXT:
               return &fb->Attachment[BUFFER_COLOR0=8  + i];
           ...
           // 注意深度模板附件返回的是深度附件
           case GL_DEPTH_STENCIL_ATTACHMENT:
              if (!_mesa_is_desktop_gl(ctx) && !_mesa_is_gles3(ctx))
                 return NULL;
              /* fall-through */
           case GL_DEPTH_ATTACHMENT_EXT:
              return &fb->Attachment[BUFFER_DEPTH];


      _mesa_framebuffer_texture(ctx, fb, attachment, att, texObj, textarget,
                                level, layer, layered);
}


void
_mesa_framebuffer_texture(struct gl_context *ctx, struct gl_framebuffer *fb,
                          GLenum attachment,
                          struct gl_renderbuffer_attachment *att,
                          struct gl_texture_object *texObj, GLenum textarget,
                          GLint level, GLuint layer, GLboolean layered)
{
   FLUSH_VERTICES(ctx, _NEW_BUFFERS);

   if (texObj) {
      if (attachment == GL_DEPTH_ATTACHMENT &&
        //作为深度附件已经附着
         reuse_framebuffer_texture_attachment(fb, BUFFER_DEPTH,
                                              BUFFER_STENCIL);
      } else if (attachment == GL_STENCIL_ATTACHMENT &&
        // 作为模板附件已经附着
         reuse_framebuffer_texture_attachment(fb, BUFFER_STENCIL,
                                              BUFFER_DEPTH);
      } else {
         //设置颜色附件之后，为该图像绑定一个renderbuffer
         set_texture_attachment(ctx, fb, att, texObj, textarget,
                                level, layer, layered);
            _mesa_update_texture_renderbuffer(ctx, fb, att);
                texImage = att->Texture->Image[att->CubeMapFace][att-
>TextureLevel];
                rb = att->Renderbuffer;
                if (!rb)
```

```
                  rb = ctx->Driver.NewRenderbuffer(ctx, ~0);
               ...

               if (driver_RenderTexture_is_safe(att))
                  ctx->Driver.RenderTexture(ctx, fb, att);
                  [jump state_tracker RenderTexture]


         if (attachment == GL_DEPTH_STENCIL_ATTACHMENT) {
            /* Above we created a new renderbuffer and attached it to the
             * depth attachment point. Now attach it to the stencil attachment
             * point too.
             */
            assert(att == &fb->Attachment[BUFFER_DEPTH]);
            reuse_framebuffer_texture_attachment(fb,BUFFER_STENCIL,
                                                 BUFFER_DEPTH);
         }
      }

      /* Set the render-to-texture flag.  We'll check this flag in
       * glTexImage() and friends to determine if we need to revalidate
       * any FBOs that might be rendering into this texture.
       * This flag never gets cleared since it's non-trivial to determine
       * when all FBOs might be done rendering to this texture.  That's OK
       * though since it's uncommon to render to a texture then repeatedly
       * call glTexImage() to change images in the texture.
       */
      texObj->_RenderToTexture = GL_TRUE;
   }
   else {
      remove_attachment(ctx, att);
      if (attachment == GL_DEPTH_STENCIL_ATTACHMENT) {
         assert(att == &fb->Attachment[BUFFER_DEPTH]);
         remove_attachment(ctx, &fb->Attachment[BUFFER_STENCIL]);
      }
   }

   invalidate_framebuffer(fb);

   simple_mtx_unlock(&fb->Mutex);
}
```

FramebufferTexture2D

```
void GLAPIENTRY
_mesa_FramebufferTexture2D(GLenum target, GLenum attachment,
                           GLenum textarget, GLuint texture, GLint level)
{
   framebuffer_texture_with_dims(2, target, attachment, textarget, texture,
                                 level, 0, "glFramebufferTexture2D");
        ...
        _mesa_framebuffer_texture(ctx, fb, attachment, att, texObj, textarget,
                                  level, layer, GL_FALSE);
}
```

- 可见该接口内部同样是调用_mesa_framebuffer_texture，而dims主要主要是内部调用前进行检查作用

将渲染缓冲附件到帧缓冲

```
_mesa_FramebufferRenderbuffer_no_error(GLenum target, GLenum attachment,
                                       GLenum renderbuffertarget,
   struct gl_framebuffer *fb = get_framebuffer_target(ctx, target);
                                       GLuint renderbuffer)

   framebuffer_renderbuffer_no_error(ctx, fb, attachment, renderbuffertarget,
                                     renderbuffer, "glFramebufferRenderbuffer");
       framebuffer_renderbuffer(ctx, fb, attachment, renderbuffertarget,
                                renderbuffer, func, true);
            ...
            _mesa_framebuffer_renderbuffer(ctx, fb, attachment, rb);

void
_mesa_framebuffer_renderbuffer(struct gl_context *ctx,
                               struct gl_framebuffer *fb,
                               GLenum attachment,
                               struct gl_renderbuffer *rb)
{

    // 设置_NEW_BUFFERS状态
   FLUSH_VERTICES(ctx, _NEW_BUFFERS);

   assert(ctx->Driver.FramebufferRenderbuffer);
   ctx->Driver.FramebufferRenderbuffer(ctx, fb, attachment, rb);
   [jump state_tracker FramebufferRenderbuffer]

   /* Some subsequent GL commands may depend on the framebuffer's visual
    * after the binding is updated.  Update visual info now.
    */
   _mesa_update_framebuffer_visual(ctx, fb);
}
```

state_tracker

rbo st_renderbuffer定义

```
/**
 * 派生的渲染缓冲类。只需要添加指向管道表面的指针。
 */
struct st_renderbuffer
{
   struct gl_renderbuffer Base;  /**< 公共渲染缓冲信息的基本结构。 */
   struct pipe_resource *texture;  /**< 指向管道资源的指针，可能表示与渲染缓冲相关的底层资
源。 */

   /* 这指向“surface_linear”或“surface_srgb”中的一个。
    * 它不保存pipe_surface的引用。 另外两个保存引用。
    */
   struct pipe_surface *surface; /**< 指向pipe_surface结构的指针，可能表示一个渲染表面。
*/
   struct pipe_surface *surface_linear; /**< 指向不同的渲染表面，可能是线性格式。 */
   struct pipe_surface *surface_srgb;   /**< 指向不同的渲染表面，可能是sRGB格式。 */

   GLboolean defined;        /**< 内容是否已定义的布尔标志。 */

   struct pipe_transfer *transfer; /**< 仅在映射资源时使用。 */

   /**
    * 仅在不支持硬件累积缓冲时使用。
    */
   boolean software;  /**< 表示是否使用软件累积缓冲的布尔值。 */
   void *data;        /**< 当映射资源时使用的数据。 */

   bool use_readpix_cache; /**< 布尔值，表示是否使用读取像素缓存。 */

   /* 从Driver.RenderTexture输入的，不要直接使用。 */
   boolean is_rtt; /**< 表示是否调用了Driver.RenderTexture。 */
   unsigned rtt_face, rtt_slice; /**< 用于渲染到纹理的相关信息。 */
   boolean rtt_layered; /**< 表示是否调用了glFramebufferTexture。 */
};
```

fbo接口

```

// st_cb_fbo.c
void
st_init_fbo_functions(struct dd_function_table *functions)
{
   functions->NewFramebuffer = _mesa_new_framebuffer;
   functions->NewRenderbuffer = st_new_renderbuffer;
   functions->FramebufferRenderbuffer = _mesa_FramebufferRenderbuffer_sw;
   functions->RenderTexture = st_render_texture;
   functions->FinishRenderTexture = st_finish_render_texture;
   functions->ValidateFramebuffer = st_validate_framebuffer;

   functions->DrawBufferAllocate = st_DrawBufferAllocate;
```

```
   functions->ReadBuffer = st_ReadBuffer;

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

创建fbo

```
struct gl_framebuffer *
_mesa_new_framebuffer(struct gl_context *ctx, GLuint name)
{
   struct gl_framebuffer *fb;
   (void) ctx;
   assert(name != 0);
   fb = CALLOC_STRUCT(gl_framebuffer);
   if (fb) {
      _mesa_initialize_user_framebuffer(fb, name);
           fb->_NumColorDrawBuffers = 1;
           fb->ColorDrawBuffer[0] = GL_COLOR_ATTACHMENT0_EXT;
           fb->_ColorDrawBufferIndexes[0] = BUFFER_COLOR0;
           fb->ColorReadBuffer = GL_COLOR_ATTACHMENT0_EXT;
           fb->_ColorReadBufferIndex = BUFFER_COLOR0;
   }
   return fb;
}
```

创建renderbuffer NewRenderbuffer

```
/**
 * Called via ctx->Driver.NewRenderbuffer()
 */
static struct gl_renderbuffer *
st_new_renderbuffer(struct gl_context *ctx, GLuint name)
{
   struct st_renderbuffer *strb = ST_CALLOC_STRUCT(st_renderbuffer);
   if (strb) {
      assert(name != 0);
      _mesa_init_renderbuffer(&strb->Base, name);
           ....
           /* The rest of these should be set later by the caller of this
function or
            * the AllocStorage method:
            */
           rb->AllocStorage = NULL;

      strb->Base.Delete = st_renderbuffer_delete;
      strb->Base.AllocStorage = st_renderbuffer_alloc_storage;
      return &strb->Base;
   }
   return NULL;
}
```

为渲染缓冲分配存储 st_renderbuffer->textur e

```
/**
 * gl_renderbuffer::AllocStorage()
 * This is called to allocate the original drawing surface, and
 * during window resize.
 */
static GLboolean
st_renderbuffer_alloc_storage(struct gl_context * ctx,
                              struct gl_renderbuffer *rb,
                              GLenum internalFormat,
                              GLuint width, GLuint height)
{
   struct st_context *st = st_context(ctx);
   struct pipe_screen *screen = st->pipe->screen;
   struct st_renderbuffer *strb = st_renderbuffer(rb);
   enum pipe_format format = PIPE_FORMAT_NONE;
   struct pipe_resource templ;
   ...

   /* Free the old surface and texture
    */
   pipe_surface_reference(&strb->surface_srgb, NULL);
   pipe_surface_reference(&strb->surface_linear, NULL);
   strb->surface = NULL;
   pipe_resource_reference(&strb->texture, NULL);

   if (rb->NumSamples > 0)
      ...
      // 选择renderbuffer格式
      format = st_choose_renderbuffer_format(st, internalFormat,samples,
samples);

   strb->Base.Format = st_pipe_format_to_mesa_format(format);

   if (width == 0 || height == 0) {
      /* if size is zero, nothing to allocate */
      return GL_TRUE;
   }

   /* Setup new texture template.
    */
   memset(&templ, 0, sizeof(templ));
   templ.target = st->internal_target;
   ...
   templ.nr_samples = rb->NumSamples;
   templ.nr_storage_samples = rb->NumStorageSamples;

   if (util_format_is_depth_or_stencil(format)) {
      templ.bind = PIPE_BIND_DEPTH_STENCIL;
   }
   else if (strb->Base.Name != 0) {
      /* this is a user-created renderbuffer */
      templ.bind = PIPE_BIND_RENDER_TARGET;
```

```
   }
   else {
      /* this is a window-system buffer */
      templ.bind = (PIPE_BIND_DISPLAY_TARGET |
                    PIPE_BIND_RENDER_TARGET);
   }

   // 存储是个纹理资源类型
   strb->texture = screen->resource_create(screen, &templ);

   if (!strb->texture)
      return FALSE;

   st_update_renderbuffer_surface(st, strb);
   [ see state_tracker RenderTexture]
   return strb->surface != NULL;
}
```

- 关于st->internal_target ，如下， 在radeonsi中为1, 故格式是PIPE_TEXTURE_2D.

```
st_create_context_priv(struct gl_context *ctx, struct pipe_context *pipe,
                       const struct st_config_options *options, bool no_error)


    ...
   /* Choose texture target for glDrawPixels, glBitmap, renderbuffers */
   if (pipe->screen->get_param(pipe->screen, PIPE_CAP_NPOT_TEXTURES))
      st->internal_target = PIPE_TEXTURE_2D;
   else
      st->internal_target = PIPE_TEXTURE_RECT;
```

*

附加Renderbuffer FramebufferRenderbuffer

```
void
_mesa_FramebufferRenderbuffer_sw(struct gl_context *ctx,
                                 struct gl_framebuffer *fb,
                                 GLenum attachment,
                                 struct gl_renderbuffer *rb)
{
   struct gl_renderbuffer_attachment *att;

   simple_mtx_lock(&fb->Mutex);

   att = get_attachment(ctx, fb, attachment, NULL);
   assert(att);
   if (rb) {
      set_renderbuffer_attachment(ctx, att, rb);
            // 将fb绑定到att->Renderbuffer
            _mesa_reference_renderbuffer(&att->Renderbuffer, rb);
      if (attachment == GL_DEPTH_STENCIL_ATTACHMENT) {
         /* do stencil attachment here (depth already done above) */
         att = get_attachment(ctx, fb, GL_STENCIL_ATTACHMENT_EXT, NULL);
```

```
         assert(att);
         set_renderbuffer_attachment(ctx, att, rb);
      }
      rb->AttachedAnytime = GL_TRUE;
   }
   else {
      remove_attachment(ctx, att);
      if (attachment == GL_DEPTH_STENCIL_ATTACHMENT) {
         /* detach stencil (depth was detached above) */
         att = get_attachment(ctx, fb, GL_STENCIL_ATTACHMENT_EXT, NULL);
         assert(att);
         remove_attachment(ctx, att);
      }
   }

   invalidate_framebuffer(fb);

   simple_mtx_unlock(&fb->Mutex);
}
```

默认fb renderbuffer的创建

```
/**
 * Allocate a renderbuffer for an on-screen window (not a user-created
 * renderbuffer).  The window system code determines the format.
 */
struct gl_renderbuffer *
st_new_renderbuffer_fb(enum pipe_format format, unsigned samples, boolean sw)
{
   struct st_renderbuffer *strb;

   strb = ST_CALLOC_STRUCT(st_renderbuffer);
   if (!strb) {
      _mesa_error(NULL, GL_OUT_OF_MEMORY, "creating renderbuffer");
      return NULL;
   }

   _mesa_init_renderbuffer(&strb->Base, 0);
   strb->Base.ClassID = 0x4242; /* just a unique value */
   strb->Base.NumSamples = samples;
   strb->Base.NumStorageSamples = samples;
   strb->Base.Format = st_pipe_format_to_mesa_format(format);
   strb->Base._BaseFormat = _mesa_get_format_base_format(strb->Base.Format);
   strb->software = sw;

   switch (format) {
   case PIPE_FORMAT_B10G10R10A2_UNORM:
   case PIPE_FORMAT_R10G10B10A2_UNORM:
      strb->Base.InternalFormat = GL_RGB10_A2;
      break;
   case PIPE_FORMAT_R10G10B10X2_UNORM:
   case PIPE_FORMAT_B10G10R10X2_UNORM:
```

```
      strb->Base.InternalFormat = GL_RGB10;
      break;
   case PIPE_FORMAT_R8G8B8A8_UNORM:
   case PIPE_FORMAT_B8G8R8A8_UNORM:
   case PIPE_FORMAT_A8R8G8B8_UNORM:
      strb->Base.InternalFormat = GL_RGBA8;
      break;
   case PIPE_FORMAT_R8G8B8X8_UNORM:
   case PIPE_FORMAT_B8G8R8X8_UNORM:
   case PIPE_FORMAT_X8R8G8B8_UNORM:
      strb->Base.InternalFormat = GL_RGB8;
      break;
   case PIPE_FORMAT_R8G8B8A8_SRGB:
   case PIPE_FORMAT_B8G8R8A8_SRGB:
   case PIPE_FORMAT_A8R8G8B8_SRGB:
      strb->Base.InternalFormat = GL_SRGB8_ALPHA8;
      break;
   case PIPE_FORMAT_R8G8B8X8_SRGB:
   case PIPE_FORMAT_B8G8R8X8_SRGB:
   case PIPE_FORMAT_X8R8G8B8_SRGB:
      strb->Base.InternalFormat = GL_SRGB8;
      break;
   case PIPE_FORMAT_B5G5R5A1_UNORM:
      strb->Base.InternalFormat = GL_RGB5_A1;
      break;
   case PIPE_FORMAT_B4G4R4A4_UNORM:
      strb->Base.InternalFormat = GL_RGBA4;
      break;
   case PIPE_FORMAT_B5G6R5_UNORM:
      strb->Base.InternalFormat = GL_RGB565;
      break;
   case PIPE_FORMAT_Z16_UNORM:
      strb->Base.InternalFormat = GL_DEPTH_COMPONENT16;
      break;
   case PIPE_FORMAT_Z32_UNORM:
      strb->Base.InternalFormat = GL_DEPTH_COMPONENT32;
      break;
   case PIPE_FORMAT_Z24_UNORM_S8_UINT:
   case PIPE_FORMAT_S8_UINT_Z24_UNORM:
      strb->Base.InternalFormat = GL_DEPTH24_STENCIL8_EXT;
      break;
   case PIPE_FORMAT_Z24X8_UNORM:
   case PIPE_FORMAT_X8Z24_UNORM:
      strb->Base.InternalFormat = GL_DEPTH_COMPONENT24;
      break;
   case PIPE_FORMAT_S8_UINT:
      strb->Base.InternalFormat = GL_STENCIL_INDEX8_EXT;
      break;
   case PIPE_FORMAT_R16G16B16A16_SNORM:
      /* accum buffer */
      strb->Base.InternalFormat = GL_RGBA16_SNORM;
      break;
   case PIPE_FORMAT_R16G16B16A16_UNORM:
      strb->Base.InternalFormat = GL_RGBA16;
      break;
   case PIPE_FORMAT_R8_UNORM:
```

```
      strb->Base.InternalFormat = GL_R8;
      break;
   case PIPE_FORMAT_R8G8_UNORM:
      strb->Base.InternalFormat = GL_RG8;
      break;
   case PIPE_FORMAT_R16_UNORM:
      strb->Base.InternalFormat = GL_R16;
      break;
   case PIPE_FORMAT_R16G16_UNORM:
      strb->Base.InternalFormat = GL_RG16;
      break;
   case PIPE_FORMAT_R32G32B32A32_FLOAT:
      strb->Base.InternalFormat = GL_RGBA32F;
      break;
   case PIPE_FORMAT_R16G16B16A16_FLOAT:
      strb->Base.InternalFormat = GL_RGBA16F;
      break;
   default:
      _mesa_problem(NULL,
                    "Unexpected format %s in st_new_renderbuffer_fb",
                    util_format_name(format));
      free(strb);
      return NULL;
   }

   /* st-specific methods */
   strb->Base.Delete = st_renderbuffer_delete;
   strb->Base.AllocStorage = st_renderbuffer_alloc_storage;

   /* surface is allocated in st_renderbuffer_alloc_storage() */
   strb->surface = NULL;

   return &strb->Base;
}
```

渲染纹理 RenderTexture

```
static void
st_render_texture(struct gl_context *ctx,
                  struct gl_framebuffer *fb,
                  struct gl_renderbuffer_attachment *att)
{
   struct st_context *st = st_context(ctx);
   struct gl_renderbuffer *rb = att->Renderbuffer;
   struct st_renderbuffer *strb = st_renderbuffer(rb);
   struct pipe_resource *pt;

   pt = get_teximage_resource(att->Texture,
                              att->CubeMapFace,
                              att->TextureLevel);

   /* point renderbuffer at texobject */
   strb->is_rtt = TRUE;
   ...
```

```
   pipe_resource_reference(&strb->texture, pt);

   st_update_renderbuffer_surface(st, strb);
   /* Invalidate buffer state so that the pipe's framebuffer state
    * gets updated.
    * That's where the new renderbuffer (which we just created) gets
    * passed to the pipe as a (color/depth) render target.
    */
   st_invalidate_buffers(st);


   /* Need to trigger a call to update_framebuffer() since we just
    * attached a new renderbuffer.
    */
   ctx->NewState |= _NEW_BUFFERS;
}

/**
 * Create or update the pipe_surface of a FBO renderbuffer.
 * This is usually called after st_finalize_texture.
 */
void
st_update_renderbuffer_surface(struct st_context *st,
                               struct st_renderbuffer *strb)
{
   struct pipe_context *pipe = st->pipe;
   struct pipe_resource *resource = strb->texture;
   const struct st_texture_object *stTexObj = NULL;
   unsigned rtt_width = strb->Base.Width;
   unsigned rtt_height = strb->Base.Height;
   unsigned rtt_depth = strb->Base.Depth;

   ...

   struct pipe_surface **psurf =
      enable_srgb ? &strb->surface_srgb : &strb->surface_linear;
   struct pipe_surface *surf = *psurf;

   if (!surf ||
       surf->texture->nr_samples != strb->Base.NumSamples ||
       surf->texture->nr_storage_samples != strb->Base.NumStorageSamples ||
       surf->format != format ||
       surf->texture != resource ||
       surf->width != rtt_width ||
       surf->height != rtt_height ||
       surf->u.tex.level != level ||
       surf->u.tex.first_layer != first_layer ||
       surf->u.tex.last_layer != last_layer) {
      /* create a new pipe_surface */
      struct pipe_surface surf_tmpl;
      memset(&surf_tmpl, 0, sizeof(surf_tmpl));
      surf_tmpl.format = format;
      surf_tmpl.u.tex.level = level;
      surf_tmpl.u.tex.first_layer = first_layer;
      surf_tmpl.u.tex.last_layer = last_layer;
```

```
      pipe_surface_release(pipe, psurf);

      *psurf = pipe->create_surface(pipe, resource, &surf_tmpl);
   }
   strb->surface = *psurf;
}
```

附件Texture Renderbuffer

默认FB分配DrawBuffer DrawBufferAllocate

```
/**
 * 通过 glDrawBuffer 调用。我们只提供这个驱动程序函数是为了检查是否需要分配新的渲染缓冲区。
 * 具体而言，当使用双缓冲视觉时，通常情况下我们不会分配前端颜色缓冲区。但是如果应用程序调用
 * glDrawBuffer(GL_FRONT)，我们需要分配该缓冲区。注意，这仅适用于窗口系统缓冲区，不适用于
 * 用户创建的 FBO。
 */
static void
st_DrawBufferAllocate(struct gl_context *ctx)
{
   struct st_context *st = st_context(ctx);
   struct gl_framebuffer *fb = ctx->DrawBuffer;

   if (_mesa_is_winsys_fbo(fb)) {
      GLuint i;
      /* 根据需要添加渲染缓冲区 */
      for (i = 0; i < fb->_NumColorDrawBuffers; i++) {
         gl_buffer_index idx = fb->_ColorDrawBufferIndexes[i];

         if (idx != BUFFER_NONE) {
            st_manager_add_color_renderbuffer(st, fb, idx);
         }
      }
   }
}

/**
 * 根据需要添加一个颜色渲染缓冲区。FBO 必须对应于窗口，而不是用户创建的 FBO。
 */
boolean
st_manager_add_color_renderbuffer(struct st_context *st,
                                  struct gl_framebuffer *fb,
                                  gl_buffer_index idx)
{
   struct st_framebuffer *stfb = st_ws_framebuffer(fb);

   /* FBO */
   if (!stfb)
      return FALSE;

   assert(_mesa_is_winsys_fbo(fb));

   if (stfb->Base.Attachment[idx].Renderbuffer)
      return TRUE;
```

```
   switch (idx) {
   case BUFFER_FRONT_LEFT:
   case BUFFER_BACK_LEFT:
   case BUFFER_FRONT_RIGHT:
   case BUFFER_BACK_RIGHT:
      break;
   default:
      return FALSE;
   }

   if (!st_framebuffer_add_renderbuffer(stfb, idx))
            rb = st_new_renderbuffer_fb(format, stfb->iface->visual->samples,
sw);
            //
            _mesa_attach_and_own_rb(&stfb->Base, idx, rb);
                fb->Attachment[bufferName].Renderbuffer = rb;
            //
            _mesa_attach_and_reference_rb(&stfb->Base, BUFFER_STENCIL, rb);
      return FALSE;

   st_framebuffer_update_attachments(stfb);

   /*
    * 强制调用状态跟踪器管理器以验证新的渲染缓冲区。
    * 可能有窗口系统渲染缓冲区可用。
    */
   if (stfb->iface)
      stfb->iface_stamp = p_atomic_read(&stfb->iface->stamp) - 1;

   st_invalidate_buffers(st);

   return TRUE;
}
```

framebuffer状态转发

处理ST_NEW_SAMPLE_STATE

多重采样设置，暂时搁置

```
/* Update the sample mask and locations for MSAA.
 */
void
st_update_sample_state(struct st_context *st)
{
   unsigned sample_mask = 0xffffffff;
   unsigned sample_count = st->state.fb_num_samples;

   if (_mesa_is_multisample_enabled(st->ctx) && sample_count > 1) {
      /* unlike in gallium/d3d10 the mask is only active if msaa is enabled */
      if (st->ctx->Multisample.SampleCoverage) {
         unsigned nr_bits = (unsigned)
            (st->ctx->Multisample.SampleCoverageValue * (float) sample_count);
```

```
         /* there's lot of ways how to do this. We just use first few bits,
          * since we have no knowledge of sample positions here. When
          * app-supplied mask though is used too might need to be smarter.
          * Also, there's an interface restriction here in theory it is
          * encouraged this mask not be the same at each pixel.
          */
         sample_mask = (1 << nr_bits) - 1;
         if (st->ctx->Multisample.SampleCoverageInvert)
            sample_mask = ~sample_mask;
      }
      if (st->ctx->Multisample.SampleMask)
         sample_mask &= st->ctx->Multisample.SampleMaskValue;
   }

   cso_set_sample_mask(st->cso_context, sample_mask);

   update_sample_locations(st);
}
```

处理_NEW_BUFFERS状态

```
void
_mesa_update_state_locked( struct gl_context *ctx )
{
   GLbitfield new_state = ctx->NewState;
   GLbitfield new_prog_state = 0x0;
   const GLbitfield computed_states = ~(_NEW_CURRENT_ATTRIB | _NEW_LINE);

   ...

   if (new_state & _NEW_BUFFERS)
      // 更新fb的draw/read framebuffers
      _mesa_update_framebuffer(ctx, ctx->ReadBuffer, ctx->DrawBuffer);
            update_framebuffer(ctx, drawFb);
            if (readFb != drawFb)
               update_framebuffer(ctx, readFb);

            _mesa_update_clamp_vertex_color(ctx, drawFb);
               ctx->Light._ClampVertexColor =
                     _mesa_get_clamp_vertex_color(ctx, drawFb);

            _mesa_update_clamp_fragment_color(ctx, drawFb);
              ctx->Color._ClampFragmentColor =
                 _mesa_get_clamp_fragment_color(ctx, drawFb);

   ...

   ctx->Driver.UpdateState(ctx);
   ctx->NewState = 0;
}
```

```c
/**
 * 更新一个 gl_framebuffer 的派生状态。
 *
 * 具体而言，更新以下 framebuffer 字段：
 *    _ColorDrawBuffers
 *    _NumColorDrawBuffers
 *    _ColorReadBuffer
 *
 * 如果 framebuffer 是用户创建的，则确保它是完整的。
 *
 * 下列函数（至少）可以影响 framebuffer 状态：
 * glReadBuffer、glDrawBuffer、glDrawBuffersARB、glFramebufferRenderbufferEXT、
 * glRenderbufferStorageEXT。
 */
static void
update_framebuffer(struct gl_context *ctx, struct gl_framebuffer *fb)
{
   if (_mesa_is_winsys_fbo(fb)) {
      /* 这是一个窗口系统的帧缓冲 */
      /* 需要更新 FB 的 GL_DRAW_BUFFER 状态以匹配上下文状态（GL_READ_BUFFER 也一样） */
      if (fb->ColorDrawBuffer[0] != ctx->Color.DrawBuffer[0]) {
         _mesa_drawbuffers(ctx, fb, ctx->Const.MaxDrawBuffers,
                           ctx->Color.DrawBuffer, NULL);
      }

      /* 如果 fb 是绑定的绘制缓冲区，则调用设备驱动程序函数。 */
      if (fb == ctx->DrawBuffer) {
         if (ctx->Driver.DrawBufferAllocate)
            ctx->Driver.DrawBufferAllocate(ctx);
            [jump state_tracker st_DrawBufferAllocate]
      }
   }
   else {
      /* 这是一个用户创建的帧缓冲。
       * 对于用户创建的帧缓冲，完整性是重要的。
       */
      if (fb->_Status != GL_FRAMEBUFFER_COMPLETE) {
         _mesa_test_framebuffer_completeness(ctx, fb);
      }
   }

   /* 严格来说，如果这个 FB 被绑定为 ctx->ReadBuffer（反之亦然，如果这个 FB 被绑定为 ctx-
>DrawBuffer），
    * 我们不需要更新绘制状态，但也没有坏处。
    */
   update_color_draw_buffers(fb);
   update_color_read_buffer(fb);

   compute_depth_max(fb);
}

/**
 * Called via ctx->Driver.UpdateState()
 */
static void
```

```
st_invalidate_state(struct gl_context *ctx)
{
   GLbitfield new_state = ctx->NewState;
   struct st_context *st = st_context(ctx);

   if (new_state & _NEW_BUFFERS) {
      st_invalidate_buffers(st);
   } else {
    ...
    }

   ...
}


void
st_invalidate_buffers(struct st_context *st)
{
   st->dirty |= ST_NEW_BLEND |
                ST_NEW_DSA |
                ST_NEW_FB_STATE |
                ST_NEW_SAMPLE_STATE |
                ST_NEW_SAMPLE_SHADING |
                ST_NEW_FS_STATE |
                ST_NEW_POLY_STIPPLE |
                ST_NEW_VIEWPORT |
                ST_NEW_RASTERIZER |
                ST_NEW_SCISSOR |
                ST_NEW_WINDOW_RECTANGLES;
}
```

- 可见在_NEW_BUFFERS状态设置后， 会触发混合，dsa,fb，次阿采样状态，fs状态，图案，视口，光栅化，窗口矩形等一些列状态更新.

处理ST_NEW_FB_STATE

这段代码用于更新帧缓冲状态（包括颜色缓冲、深度缓冲、模板缓冲等）。以下是对该代码的翻译注释：

```c
/**
 * 更新帧缓冲状态（颜色、深度、模板缓冲等）
 */
void
st_update_framebuffer_state(struct st_context *st)
{
   struct pipe_framebuffer_state framebuffer;
   struct gl_framebuffer *fb = st->ctx->DrawBuffer;
   struct st_renderbuffer *strb;
   GLuint i;

   // 刷新位图缓存和失效读取像素缓存
   st_flush_bitmap_cache(st);
```

```
   st_invalidate_readpix_cache(st);

   // 更新帧缓冲的方向
   st->state.fb_orientation = st_fb_orientation(fb);

   /**
    * 量化派生的默认采样数：
    *
    * 通过查询驱动程序支持的多重采样值，对应用程序请求的采样数 NumSamples 进行合法化。
    * 详见提交记录 eb9cf3c 以获取更多信息。
    */
   fb->DefaultGeometry._NumSamples =
      framebuffer_quantize_num_samples(st, fb->DefaultGeometry.NumSamples);

   // 设置帧缓冲的宽度、高度、采样数、层数
   framebuffer.width  = _mesa_geometric_width(fb);
   framebuffer.height = _mesa_geometric_height(fb);
   framebuffer.samples = _mesa_geometric_samples(fb);
   framebuffer.layers = _mesa_geometric_layers(fb);

   /* 检查 Mesa 的 ctx->DrawBuffer->_ColorDrawBuffers 状态，
    * 以确定要绘制到哪些表面
    */
    // fb 初始化为1
   framebuffer.nr_cbufs = fb->_NumColorDrawBuffers;

   for (i = 0; i < fb->_NumColorDrawBuffers; i++) {
      framebuffer.cbufs[i] = NULL;
      strb = st_renderbuffer(fb->_ColorDrawBuffers[i]);

      if (strb) {
         if (strb->is_rtt || (strb->texture &&
             _mesa_get_format_color_encoding(strb->Base.Format) == GL_SRGB)) {
            /* 渲染到 GL 纹理，可能需要更新表面 */
            st_update_renderbuffer_surface(st, strb);
         }

         if (strb->surface) {
            framebuffer.cbufs[i] = strb->surface;
            update_framebuffer_size(&framebuffer, strb->surface);
         }
         strb->defined = GL_TRUE; /* 我们将绘制一些东西 */
      }
   }

   for (i = framebuffer.nr_cbufs; i < PIPE_MAX_COLOR_BUFS; i++) {
      framebuffer.cbufs[i] = NULL;
   }

   /* 删除末尾的 GL_NONE 绘制缓冲区。*/
   while (framebuffer.nr_cbufs &&
          !framebuffer.cbufs[framebuffer.nr_cbufs - 1]) {
      framebuffer.nr_cbufs--;
   }

   /*
```

```
     * 深度/模板渲染缓冲区/表面。
     */
    strb = st_renderbuffer(fb->Attachment[BUFFER_DEPTH].Renderbuffer);
    if (!strb)
       strb = st_renderbuffer(fb->Attachment[BUFFER_STENCIL].Renderbuffer);

    if (strb) {
       if (strb->is_rtt) {
          /* 渲染到 GL 纹理，可能需要更新表面 */
          st_update_renderbuffer_surface(st, strb);
       }
       framebuffer.zsbuf = strb->surface;
       if (strb->surface)
          update_framebuffer_size(&framebuffer, strb->surface);
    }
    else
       framebuffer.zsbuf = NULL;

    if (framebuffer.width == USHRT_MAX)
       framebuffer.width = 0;
    if (framebuffer.height == USHRT_MAX)
       framebuffer.height = 0;

    // 设置管线对象的帧缓冲状态
    cso_set_framebuffer(st->cso_context, &framebuffer);

    st->state.fb_width = framebuffer.width;
    st->state.fb_height = framebuffer.height;
    st->state.fb_num_samples = util_framebuffer_get_num_samples(&framebuffer);
    st->state.fb_num_layers = util_framebuffer_get_num_layers(&framebuffer);
    st->state.fb_num_cb = framebuffer.nr_cbufs;
}
```

fbo颜色drawbuffers的更新

```
/**
 * Update the (derived) list of color drawing renderbuffer pointers.
 * Later, when we're rendering we'll loop from 0 to _NumColorDrawBuffers
 * writing colors.
 */
static void
update_color_draw_buffers(struct gl_framebuffer *fb)
{
   GLuint output;

   /* set 0th buffer to NULL now in case _NumColorDrawBuffers is zero */
   fb->_ColorDrawBuffers[0] = NULL;

   for (output = 0; output < fb->_NumColorDrawBuffers; output++) {
      gl_buffer_index buf = fb->_ColorDrawBufferIndexes[output];
      if (buf != BUFFER_NONE) {
         fb->_ColorDrawBuffers[output] = fb->Attachment[buf].Renderbuffer;
      }
      else {
         fb->_ColorDrawBuffers[output] = NULL;
```

```
      }
   }
}
```

fbo颜色readbuffers的更新

```
/**
 * Update the (derived) color read renderbuffer pointer.
 * Unlike the DrawBuffer, we can only read from one (or zero) color buffers.
 */
static void
update_color_read_buffer(struct gl_framebuffer *fb)
{
   if (fb->_ColorReadBufferIndex == BUFFER_NONE ||
       fb->DeletePending ||
       fb->Width == 0 ||
       fb->Height == 0) {
      fb->_ColorReadBuffer = NULL; /* legal! */
   }
   else {
      assert(fb->_ColorReadBufferIndex >= 0);
      assert(fb->_ColorReadBufferIndex < BUFFER_COUNT);
      fb->_ColorReadBuffer
         = fb->Attachment[fb->_ColorReadBufferIndex].Renderbuffer;
   }
}
```

设置驱动framebuffer cso_set_framebuffer

```
void cso_set_framebuffer(struct cso_context *ctx,
                         const struct pipe_framebuffer_state *fb)
{
   if (memcmp(&ctx->fb, fb, sizeof(*fb)) != 0) {
      util_copy_framebuffer_state(&ctx->fb, fb);
      ctx->pipe->set_framebuffer_state(ctx->pipe, fb);
      [jump radeonsi si_set_framebuffer_state]
   }
}
```

RadeonSI

fb状态设置接口及发射函数

```
    sctx->b.set_framebuffer_state = si_set_framebuffer_state;

    sctx->atoms.s.framebuffer.emit = si_emit_framebuffer_state;
```

帧缓冲状态设置

```
static void si_set_framebuffer_state(struct pipe_context *ctx,
                                     const struct pipe_framebuffer_state *state)
{
   struct si_context *sctx = (struct si_context *)ctx;
   struct si_surface *surf = NULL;
   struct si_texture *tex;
   bool old_any_dst_linear = sctx->framebuffer.any_dst_linear;
   unsigned old_nr_samples = sctx->framebuffer.nr_samples;
   unsigned old_colorbuf_enabled_4bit = sctx->framebuffer.colorbuf_enabled_4bit;
   bool old_has_zsbuf = !!sctx->framebuffer.state.zsbuf;
   bool old_has_stencil =
       old_has_zsbuf &&
       ((struct si_texture *)sctx->framebuffer.state.zsbuf->texture)->surface.has_stencil;
   bool unbound = false;
   int i;

   /* 拒绝零大小的帧缓冲，这是因为 SI 存在一个硬件 bug，当 PA_SU_HARDWARE_SCREEN_OFFSET != 0 且
    * any_scissor.BR_X/Y <= 0 时会发生。我们可以在这里实现完整的解决方案，但这是一个无用的情况。
    */
   if ((!state->width || !state->height) && (state->nr_cbufs || state->zsbuf)) {
      unreachable("the framebuffer shouldn't have zero area");
      return;
   }

   si_update_fb_dirtiness_after_rendering(sctx);

   // 处理Dcc
   ...

   /* 仅在更改帧缓冲区状态时刷新 TC，因为唯一可以更改纹理的客户端不使用 TC 的是帧缓冲区。 */
   if (sctx->framebuffer.uncompressed_cb_mask)
      si_make_CB_shader_coherent(sctx, sctx->framebuffer.nr_samples,
                                 sctx->framebuffer.CB_has_shader_readable_metadata);

   sctx->flags |= SI_CONTEXT_CS_PARTIAL_FLUSH;

   /* u_blitter 在连续进行多个 blit 时不会调用深度解压缩，但 DB 的唯一情况是在进行 generate_mipmap
    * 时。因此，在这里我们在单独的 generate_mipmap blit 之间手动刷新 DB。
    * 注意，较低的 mipmap 级别不会被压缩。
    */
   if (sctx->generate_mipmap_for_depth) {
      si_make_DB_shader_coherent(sctx, 1, false,
                                 sctx->framebuffer.DB_has_shader_readable_metadata);
   } else if (sctx->chip_class == GFX9) {
      /* 似乎 DB 元数据会在以下序列中“泄漏”：
       *   - 深度清除
       *   - 为着色器图像写 DCC 解压缩（禁用 DB 的情况下）
```

```
         *  - 使用 DEPTH_BEFORE_SHADER=1 渲染
         * 刷新 DB 元数据可以解决此问题。
         */
        sctx->flags |= SI_CONTEXT_FLUSH_AND_INV_DB_META;
    }

    /* 取旧和新计数的最大值。如果新计数较低，则需要脏化以禁用未绑定的颜色缓冲区。 */
    sctx->framebuffer.dirty_cbufs |=
        (1 << MAX2(sctx->framebuffer.state.nr_cbufs, state->nr_cbufs)) - 1;
    sctx->framebuffer.dirty_zsbuf |= sctx->framebuffer.state.zsbuf != state-
>zsbuf;

    si_dec_framebuffer_counters(&sctx->framebuffer.state);
    util_copy_framebuffer_state(&sctx->framebuffer.state, state);

    sctx->framebuffer.colorbuf_enabled_4bit = 0;
    sctx->framebuffer.spi_shader_col_format = 0;
    sctx->framebuffer.spi_shader_col_format_alpha = 0;
    sctx->framebuffer.spi_shader_col_format_blend = 0;
    sctx->framebuffer.spi_shader_col_format_blend
```

```
static void si_set_framebuffer_state(struct pipe_context *ctx,
                     const struct pipe_framebuffer_state *state)
{
    struct si_context *sctx = (struct si_context *)ctx;
    struct si_surface *surf = NULL;
    struct si_texture *tex;
    bool old_any_dst_linear = sctx->framebuffer.any_dst_linear;
    unsigned old_nr_samples = sctx->framebuffer.nr_samples;
    unsigned old_colorbuf_enabled_4bit = sctx->framebuffer.colorbuf_enabled_4bit;
    bool old_has_zsbuf = !!sctx->framebuffer.state.zsbuf;
    bool old_has_stencil =
        old_has_zsbuf &&
        ((struct si_texture*)sctx->framebuffer.state.zsbuf->texture)-
>surface.has_stencil;
    bool unbound = false;
    int i;

    /* Reject zero-sized framebuffers due to a hw bug on SI that occurs
     * when PA_SU_HARDWARE_SCREEN_OFFSET != 0 and any_scissor.BR_X/Y <= 0.
     * We could implement the full workaround here, but it's a useless case.
     */
    if ((!state->width || !state->height) && (state->nr_cbufs || state->zsbuf)) {
        unreachable("the framebuffer shouldn't have zero area");
        return;
    }

    si_update_fb_dirtiness_after_rendering(sctx);

    // 设置dcc处理标志
    surf->dcc_incompatible = false/true;
    ...
```

```
    //设置amdgpu tc l2标志
    if (sctx->framebuffer.uncompressed_cb_mask)
        si_make_CB_shader_coherent(sctx, sctx->framebuffer.nr_samples,
                        sctx->framebuffer.CB_has_shader_readable_metadata);

    sctx->flags |= SI_CONTEXT_CS_PARTIAL_FLUSH;

    /* u_blitter doesn't invoke depth decompression when it does multiple
     * blits in a row, but the only case when it matters for DB is when
     * doing generate_mipmap. So here we flush DB manually between
     * individual generate_mipmap blits.
     * Note that lower mipmap levels aren't compressed.
     */
    if (sctx->generate_mipmap_for_depth) {
        si_make_DB_shader_coherent(sctx, 1, false,
                        sctx->framebuffer.DB_has_shader_readable_metadata);
    } else if (sctx->chip_class == GFX9) {
        ...
    }

    /* Take the maximum of the old and new count. If the new count is lower,
     * dirtying is needed to disable the unbound colorbuffers.
     */
    sctx->framebuffer.dirty_cbufs |=
        (1 << MAX2(sctx->framebuffer.state.nr_cbufs, state->nr_cbufs)) - 1;
    sctx->framebuffer.dirty_zsbuf |= sctx->framebuffer.state.zsbuf != state-
>zsbuf;

    si_dec_framebuffer_counters(&sctx->framebuffer.state);
    util_copy_framebuffer_state(&sctx->framebuffer.state, state);

    sctx->framebuffer.colorbuf_enabled_4bit = 0;
    sctx->framebuffer.xxx = xxx;
    ...
    sctx->framebuffer.nr_samples = util_framebuffer_get_num_samples(state);
    sctx->framebuffer.nr_color_samples = sctx->framebuffer.nr_samples;
    sctx->framebuffer.log_samples = util_logbase2(sctx->framebuffer.nr_samples);


    unsigned num_bpp64_colorbufs = 0;

    for (i = 0; i < state->nr_cbufs; i++) {
        if (!state->cbufs[i])
            continue;

        surf = (struct si_surface*)state->cbufs[i];
        tex = (struct si_texture*)surf->base.texture;

        if (!surf->color_initialized) {
            si_initialize_color_surface(sctx, surf);
        }

        sctx->framebuffer.colorbuf_enabled_4bit |= 0xf << (i * 4);

        if (tex->surface.is_linear)
```

```
            sctx->framebuffer.any_dst_linear = true;
        if (tex->surface.bpe >= 8)
            num_bpp64_colorbufs++;

        si_context_add_resource_size(sctx, surf->base.texture);

        ...

    }

    struct si_texture *zstex = NULL;

    if (state->zsbuf) {
        surf = (struct si_surface*)state->zsbuf;
        zstex = (struct si_texture*)surf->base.texture;

        if (!surf->depth_initialized) {
            si_init_depth_surface(sctx, surf);
        }
        ...
        si_context_add_resource_size(sctx, surf->base.texture);
    }

    // for SI_PS_IMAGE_COLORBUF0
    si_update_ps_colorbuf0_slot(sctx);

    si_update_poly_offset_state(sctx);
    si_mark_atom_dirty(sctx, &sctx->atoms.s.cb_render_state);
    si_mark_atom_dirty(sctx, &sctx->atoms.s.framebuffer);

    if (sctx->framebuffer.any_dst_linear != old_any_dst_linear)
        si_mark_atom_dirty(sctx, &sctx->atoms.s.msaa_config);

    if (sctx->framebuffer.nr_samples != old_nr_samples) {
        struct pipe_constant_buffer constbuf = {0};

        si_mark_atom_dirty(sctx, &sctx->atoms.s.msaa_config);
        si_mark_atom_dirty(sctx, &sctx->atoms.s.db_render_state);

        constbuf.buffer = sctx->sample_pos_buffer;

        /* Set sample locations as fragment shader constants. */
        switch (sctx->framebuffer.nr_samples) {
        case 1:
            break;
        ...
        case 16:
            constbuf.buffer_offset = (ubyte*)sctx->sample_positions.x16 -
                         (ubyte*)sctx->sample_positions.x1;
            break;
        default:
            PRINT_ERR("Requested an invalid number of samples %i.\n",
                 sctx->framebuffer.nr_samples);
            assert(0);
        }
        constbuf.buffer_size = sctx->framebuffer.nr_samples * 2 * 4;
```

```
        si_set_rw_buffer(sctx, SI_PS_CONST_SAMPLE_POSITIONS, &constbuf);

        si_mark_atom_dirty(sctx, &sctx->atoms.s.msaa_sample_locs);
    }

    sctx->do_update_shaders = true;

    if (!sctx->decompression_enabled) {
        /* Prevent textures decompression when the framebuffer state
         * changes come from the decompression passes themselves.
         */
        sctx->need_check_render_feedback = true;
    }
}
```

更新RW buffer的colorbuf0 slot SI_PS_IMAGE_COLORBUF0 (for KHR_blend_equation_advanced);

```
void si_update_ps_colorbuf0_slot(struct si_context *sctx)
{
    struct si_buffer_resources *buffers = &sctx->rw_buffers;
    struct si_descriptors *descs = &sctx->descriptors[SI_DESCS_RW_BUFFERS];
    unsigned slot = SI_PS_IMAGE_COLORBUF0;
    struct pipe_surface *surf = NULL;

    /* si_texture_disable_dcc can get us here again. */
    if (sctx->blitter->running)
        return;

    /* See whether FBFETCH is used and color buffer 0 is set. */
    // 使用FBFETCH 扩展时使用
    if (sctx->ps_shader.cso &&
        sctx->ps_shader.cso->info.opcode_count[TGSI_OPCODE_FBFETCH] &&
        sctx->framebuffer.state.nr_cbufs &&
        sctx->framebuffer.state.cbufs[0])
        surf = sctx->framebuffer.state.cbufs[0];

    /* Return if FBFETCH transitions from disabled to disabled. */
    if (!buffers->buffers[slot] && !surf)
        return;

    sctx->ps_uses_fbfetch = surf != NULL;
    si_update_ps_iter_samples(sctx);

    if (surf) {
        struct si_texture *tex = (struct si_texture*)surf->texture;
        struct pipe_image_view view;

        assert(tex);
        assert(!tex->is_depth);
```

```
        /* Disable DCC, because the texture is used as both a sampler
         * and color buffer.
         */
        si_texture_disable_dcc(sctx, tex);

        if (tex->buffer.b.b.nr_samples <= 1 && tex->cmask_buffer) {
            /* Disable CMASK. */
            assert(tex->cmask_buffer != &tex->buffer);
            si_eliminate_fast_color_clear(sctx, tex);
            si_texture_discard_cmask(sctx->screen, tex);
        }

        view.resource = surf->texture;
        view.format = surf->format;
        view.access = PIPE_IMAGE_ACCESS_READ;
        view.u.tex.first_layer = surf->u.tex.first_layer;
        view.u.tex.last_layer = surf->u.tex.last_layer;
        view.u.tex.level = surf->u.tex.level;

        /* Set the descriptor. */
        uint32_t *desc = descs->list + slot*4;
        memset(desc, 0, 16 * 4);
        si_set_shader_image_desc(sctx, &view, true, desc, desc + 8);

        pipe_resource_reference(&buffers->buffers[slot], &tex->buffer.b.b);
        radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                      &tex->buffer, RADEON_USAGE_READ,
                      RADEON_PRIO_SHADER_RW_IMAGE);
        buffers->enabled_mask |= 1u << slot;
    } else {
        /* Clear the descriptor. */
        memset(descs->list + slot*4, 0, 8*4);
        pipe_resource_reference(&buffers->buffers[slot], NULL);
        buffers->enabled_mask &= ~(1u << slot);
    }

    sctx->descriptors_dirty |= 1u << SI_DESCS_RW_BUFFERS;
}
```

###

```
/*
 * 推断帧缓冲区和光栅化器之间的状态
 */
static void si_update_poly_offset_state(struct si_context *sctx)
{
   struct si_state_rasterizer *rs = sctx->queued.named.rasterizer;

   if (!rs || !rs->uses_poly_offset || !sctx->framebuffer.state.zsbuf) {
      si_pm4_bind_state(sctx, poly_offset, NULL);
      return;
   }

   /* 使用用户格式，而不是 db_render_format，以便多边形偏移符合应用程序的预期。 */
   switch (sctx->framebuffer.state.zsbuf->texture->format) {
```

```
      case PIPE_FORMAT_Z16_UNORM:
         si_pm4_bind_state(sctx, poly_offset, &rs->pm4_poly_offset[0]);
         break;
      default: /* 24-bit */
         si_pm4_bind_state(sctx, poly_offset, &rs->pm4_poly_offset[1]);
         break;
      case PIPE_FORMAT_Z32_FLOAT:
      case PIPE_FORMAT_Z32_FLOAT_S8X24_UINT:
         si_pm4_bind_state(sctx, poly_offset, &rs->pm4_poly_offset[2]);
         break;
   }
}
```

帧缓冲状态信息下发 si_emit_framebuffer_state

在启用一个新的gfx_cs时会设置脏位s.framebuffer.emit，

```
static void si_emit_framebuffer_state(struct si_context *sctx)
{
    struct radeon_cmdbuf *cs = sctx->gfx_cs;
    struct pipe_framebuffer_state *state = &sctx->framebuffer.state;
    unsigned i, nr_cbufs = state->nr_cbufs;
    struct si_texture *tex = NULL;
    struct si_surface *cb = NULL;
    unsigned cb_color_info = 0;

    /* Colorbuffers. */
    for (i = 0; i < nr_cbufs; i++) {
        uint64_t cb_color_base, cb_color_fmask, cb_color_cmask, cb_dcc_base;
        unsigned cb_color_attrib;

        if (!(sctx->framebuffer.dirty_cbufs & (1 << i)))
            continue;

        cb = (struct si_surface*)state->cbufs[i];
        if (!cb) {
            radeon_set_context_reg(cs, R_028C70_CB_COLOR0_INFO + i * 0x3C,
                                   S_028C70_FORMAT(V_028C70_COLOR_INVALID));
            continue;
        }

        tex = (struct si_texture *)cb->base.texture;
        radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                      &tex->buffer, RADEON_USAGE_READWRITE,
                      tex->buffer.b.b.nr_samples > 1 ?
                          RADEON_PRIO_COLOR_BUFFER_MSAA :
                          RADEON_PRIO_COLOR_BUFFER);
        // 处理dcc,cmask
        ...

        /* Compute mutable surface parameters. */
        cb_color_base = tex->buffer.gpu_address >> 8;
        cb_color_fmask = 0;
        cb_color_cmask = tex->cmask_base_address_reg;
```

```
        cb_dcc_base = 0;
        cb_color_info = cb->cb_color_info | tex->cb_color_info;
        cb_color_attrib = cb->cb_color_attrib;

        if (cb->base.u.tex.level > 0)
            cb_color_info &= C_028C70_FAST_CLEAR;

        if (vi_dcc_enabled(tex, cb->base.u.tex.level)) {
            ...
        }

        if (sctx->chip_class >= GFX9) {
            ...
        } else {
            /* Compute mutable surface parameters (SI-CI-VI). */
            const struct legacy_surf_level *level_info =
                &tex->surface.u.legacy.level[cb->base.u.tex.level];
            unsigned pitch_tile_max, slice_tile_max, tile_mode_index;
            unsigned cb_color_pitch, cb_color_slice, cb_color_fmask_slice;

            cb_color_base += level_info->offset >> 8;
            /* Only macrotiled modes can set tile swizzle. */
            if (level_info->mode == RADEON_SURF_MODE_2D)
                cb_color_base |= tex->surface.tile_swizzle;

            if (!tex->surface.fmask_size)
                cb_color_fmask = cb_color_base;
            if (cb->base.u.tex.level > 0)
                cb_color_cmask = cb_color_base;
            if (cb_dcc_base)
                cb_dcc_base += level_info->dcc_offset >> 8;

            pitch_tile_max = level_info->nblk_x / 8 - 1;
            slice_tile_max = level_info->nblk_x *
                     level_info->nblk_y / 64 - 1;
            tile_mode_index = si_tile_mode_index(tex, cb->base.u.tex.level,
false);

            cb_color_attrib |= S_028C74_TILE_MODE_INDEX(tile_mode_index);
            cb_color_pitch = S_028C64_TILE_MAX(pitch_tile_max);
            cb_color_slice = S_028C68_TILE_MAX(slice_tile_max);

            if (tex->surface.fmask_size) {
                ...
            } else {
                /* This must be set for fast clear to work without FMASK. */
                if (sctx->chip_class >= CIK)
                    cb_color_pitch |= S_028C64_FMASK_TILE_MAX(pitch_tile_max);
                cb_color_attrib |=
S_028C74_FMASK_TILE_MODE_INDEX(tile_mode_index);
                cb_color_fmask_slice = S_028C88_TILE_MAX(slice_tile_max);
            }

            radeon_set_context_reg_seq(cs, R_028C60_CB_COLOR0_BASE + i * 0x3C,
                           sctx->chip_class >= VI ? 14 : 13);
            radeon_emit(cs, cb_color_base);     /* CB_COLOR0_BASE */
```

```
            radeon_emit(cs, cb_color_pitch);    /* CB_COLOR0_PITCH */
            radeon_emit(cs, cb_color_slice);    /* CB_COLOR0_SLICE */
            radeon_emit(cs, cb->cb_color_view); /* CB_COLOR0_VIEW */
            radeon_emit(cs, cb_color_info);     /* CB_COLOR0_INFO */
            radeon_emit(cs, cb_color_attrib);   /* CB_COLOR0_ATTRIB */
            radeon_emit(cs, cb->cb_dcc_control);    /* CB_COLOR0_DCC_CONTROL */
            radeon_emit(cs, cb_color_cmask);    /* CB_COLOR0_CMASK */
            radeon_emit(cs, tex->surface.u.legacy.cmask_slice_tile_max); /*
CB_COLOR0_CMASK_SLICE */
            radeon_emit(cs, cb_color_fmask);        /* CB_COLOR0_FMASK */
            radeon_emit(cs, cb_color_fmask_slice);      /* CB_COLOR0_FMASK_SLICE
*/
            radeon_emit(cs, tex->color_clear_value[0]); /* CB_COLOR0_CLEAR_WORD0
*/
            radeon_emit(cs, tex->color_clear_value[1]); /* CB_COLOR0_CLEAR_WORD1
*/

            if (sctx->chip_class >= VI) /* R_028C94_CB_COLOR0_DCC_BASE */
                radeon_emit(cs, cb_dcc_base);
        }
    }

    for (; i < 8 ; i++)
        if (sctx->framebuffer.dirty_cbufs & (1 << i))
            radeon_set_context_reg(cs, R_028C70_CB_COLOR0_INFO + i * 0x3C, 0);

    /* ZS buffer. */
    if (state->zsbuf && sctx->framebuffer.dirty_zsbuf) {
        struct si_surface *zb = (struct si_surface*)state->zsbuf;
        struct si_texture *tex = (struct si_texture*)zb->base.texture;

        radeon_add_to_buffer_list(sctx, sctx->gfx_cs,
                      &tex->buffer, RADEON_USAGE_READWRITE,
                      zb->base.texture->nr_samples > 1 ?
                          RADEON_PRIO_DEPTH_BUFFER_MSAA :
                          RADEON_PRIO_DEPTH_BUFFER);

        if (sctx->chip_class >= GFX9) {
            ...
        } else {
            radeon_set_context_reg(cs, R_028014_DB_HTILE_DATA_BASE, zb-
>db_htile_data_base);

            radeon_set_context_reg_seq(cs, R_02803C_DB_DEPTH_INFO, 9);
            radeon_emit(cs, zb->db_depth_info); /* DB_DEPTH_INFO */
            radeon_emit(cs, zb->db_z_info |     /* DB_Z_INFO */
                    S_028040_ZRANGE_PRECISION(tex->depth_clear_value != 0));
            radeon_emit(cs, zb->db_stencil_info);   /* DB_STENCIL_INFO */
            radeon_emit(cs, zb->db_depth_base); /* DB_Z_READ_BASE */
            radeon_emit(cs, zb->db_stencil_base);   /* DB_STENCIL_READ_BASE */
            radeon_emit(cs, zb->db_depth_base); /* DB_Z_WRITE_BASE */
            radeon_emit(cs, zb->db_stencil_base);   /* DB_STENCIL_WRITE_BASE */
            radeon_emit(cs, zb->db_depth_size); /* DB_DEPTH_SIZE */
            radeon_emit(cs, zb->db_depth_slice);    /* DB_DEPTH_SLICE */
        }
```

```
        radeon_set_context_reg_seq(cs, R_028028_DB_STENCIL_CLEAR, 2);
        radeon_emit(cs, tex->stencil_clear_value); /* R_028028_DB_STENCIL_CLEAR
*/
        radeon_emit(cs, fui(tex->depth_clear_value)); /* R_02802C_DB_DEPTH_CLEAR
*/

        radeon_set_context_reg(cs, R_028008_DB_DEPTH_VIEW, zb->db_depth_view);
        radeon_set_context_reg(cs, R_028ABC_DB_HTILE_SURFACE, zb-
>db_htile_surface);
    } else if (sctx->framebuffer.dirty_zsbuf) {
        if (sctx->chip_class >= GFX9)
            radeon_set_context_reg_seq(cs, R_028038_DB_Z_INFO, 2);
        else
            radeon_set_context_reg_seq(cs, R_028040_DB_Z_INFO, 2);

        radeon_emit(cs, S_028040_FORMAT(V_028040_Z_INVALID)); /* DB_Z_INFO */
        radeon_emit(cs, S_028044_FORMAT(V_028044_STENCIL_INVALID)); /*
DB_STENCIL_INFO */
    }

    /* Framebuffer dimensions. */
        /* PA_SC_WINDOW_SCISSOR_TL is set in si_init_config() */
    radeon_set_context_reg(cs, R_028208_PA_SC_WINDOW_SCISSOR_BR,
                S_028208_BR_X(state->width) | S_028208_BR_Y(state->height));

    if (sctx->screen->dfsm_allowed) {
        radeon_emit(cs, PKT3(PKT3_EVENT_WRITE, 0, 0));
        radeon_emit(cs, EVENT_TYPE(V_028A90_BREAK_BATCH) | EVENT_INDEX(0));
    }

    sctx->framebuffer.dirty_cbufs = 0;
    sctx->framebuffer.dirty_zsbuf = false;
}
```

测试

使用piglit的gl-3.0-render-integer 测试

```
c00e6900 SET_CONTEXT_REG:
00000318
01000e00         CB_COLOR0_BASE <- 0x01000e00
01800018         CB_COLOR0_PITCH <- TILE_MAX = 24 (0x18)
                                    FMASK_TILE_MAX = 24 (0x18)
00000270         CB_COLOR0_SLICE <- TILE_MAX = 624 (0x00270)
00000000         CB_COLOR0_VIEW <- SLICE_START = 0
                                   SLICE_MAX = 0
00070538         CB_COLOR0_INFO <- ENDIAN = ENDIAN_NONE
                                   FORMAT = COLOR_32_32_32_32
                                   LINEAR_GENERAL = 0
                                   NUMBER_TYPE = NUMBER_SINT
                                   COMP_SWAP = SWAP_STD
                                   FAST_CLEAR = 0
                                   COMPRESSION = 0
```

```
                                    BLEND_CLAMP = 0
                                    BLEND_BYPASS = 1
                                    SIMPLE_FLOAT = 1
                                    ROUND_MODE = 1
                                    CMASK_IS_LINEAR = 0
                                    BLEND_OPT_DONT_RD_DST = FORCE_OPT_AUTO
                                    BLEND_OPT_DISCARD_PIXEL = FORCE_OPT_AUTO
                                    FMASK_COMPRESSION_DISABLE = 0
                                    FMASK_COMPRESS_1FRAG_ONLY = 0
                                    DCC_ENABLE = 0
                                    CMASK_ADDR_TYPE = 0
000001ad         CB_COLOR0_ATTRIB <- TILE_MODE_INDEX = 13 (0xd)
                                      FMASK_TILE_MODE_INDEX = 13 (0xd)
                                      FMASK_BANK_HEIGHT = 0
                                      NUM_SAMPLES = 0
                                      NUM_FRAGMENTS = 0
                                      FORCE_DST_ALPHA_1 = 0
00000208         CB_COLOR0_DCC_CONTROL <- OVERWRITE_COMBINER_DISABLE = 0
                                          KEY_CLEAR_ENABLE = 0
                                          MAX_UNCOMPRESSED_BLOCK_SIZE =
MAX_BLOCK_SIZE_256B
                                          MIN_COMPRESSED_BLOCK_SIZE =
MIN_BLOCK_SIZE_32B
                                          MAX_COMPRESSED_BLOCK_SIZE = 0
                                          COLOR_TRANSFORM = 0
                                          INDEPENDENT_64B_BLOCKS = 1
                                          LOSSY_RGB_PRECISION = 0
                                          LOSSY_ALPHA_PRECISION = 0
01000e00         CB_COLOR0_CMASK <- 0x01000e00
00000003         CB_COLOR0_CMASK_SLICE <- TILE_MAX = 3
01000e00         CB_COLOR0_FMASK <- 0x01000e00
00000270         CB_COLOR0_FMASK_SLICE <- TILE_MAX = 624 (0x00270)
00000000         CB_COLOR0_CLEAR_WORD0 <- 0
00000000         CB_COLOR0_CLEAR_WORD1 <- 0
00000000         CB_COLOR0_DCC_BASE <- 0
```

- 通过日志可得出纹理buffer地址，不过由于这个是右移8位的地址，所以真实地址为0x01000e0000

正好与

```
mesa: glTexImage2D GL_TEXTURE_2D 0 GL_RGBA32I 200 200 1 0 GL_RGBA_INTEGER
GL_UNSIGNED_BYTE (nil)
 fb 0x55fb8db28420 output 0
ST_NEW_SAMPLERS 0
st-dirty ST_NEW_SAMPLERS 288230376152727552
VM start=0x1000E0000  end=0x10017D000 | Texture 200x200x1, 1 levels, 1 samples,
r32g32b32a32_sint
```

输出地址一致

附录 寄存器

CB:CB_COLOR_CONTROL

CB:CB_COLOR_CONTROL 是一个可读写的 32 位寄存器，用于控制通用的颜色缓冲（CB）行为，适用于所有多渲染目标（MRTs）。以下是字段的定义：

字段名称	位范围	默认值	描述
DEGAMMA_ENABLE	3	none	如果为真，则每个 UNORM 格式的 COLOR_8_8_8_8 或 COLOR_8 MRT 将被视为 SRGB 格式。这会影响正常绘制和解析操作。此位存在是为了与旧架构兼容，旧架构不具有 SRGB 数字类型。
MODE	6:4	none	该字段选择标准颜色处理或几种主要操作模式。
			可能的值：
			00 - CB_DISABLE: 禁用对颜色缓冲的绘制。导致 DB 不会将瓦片/四块发送到 CB。CB 本身会忽略此字段。
			01 - CB_NORMAL: 正常渲染模式。DB 应该发送用于像素导出的瓦片和四块。
			02 - CB_ELIMINATE_FAST_CLEAR: 使用清除颜色填充已经快速清除的颜色表面位置。DB 应只发送瓦片。
			03 - CB_RESOLVE: 从 MRT0 读取，对所有样本求平均，并写入 MRT1，它是单样本的。DB 应该只发送瓦片。
			04 - 保留
			05 - CB_FMASK_DECOMPRESS: 解压 FMASK 缓冲区以获得可读取的纹理格式。在执行此操作之前不需要 CB_ELIMINATE_FAST_CLEAR 通道。DB 应只发送瓦片。
ROP3	23:16	none	此字段支持 28 个布尔操作，将源和目标或刷子和目标组合在一起，其中刷子由着色器提供，以替换源。代码 0xCC (11001100) 将源复制到目标，从而禁用 ROP 功能。如果任何 MRT 启用混合，必须禁用 ROP。
			可能的值：
			00 - 0x00: BLACKNESS
			05 - 0x05
			10 - 0x0A

字段名称	位范围	默认值	描述
			15 - 0x0F
			17 - 0x11: NOTSRCERASE
			34 - 0x22
			51 - 0x33: NOTSRCCOPY
			68 - 0x44: SRCERASE
			80 - 0x50
			85 - 0x55: DSTINVERT
			90 - 0x5A: PATINVERT
			95 - 0x5F
			102 - 0x66: SRCINVERT
			119 - 0x77
			136 - 0x88: SRCAND
			153 - 0x99
			160 - 0xA0
			165 - 0xA5
			170 - 0xAA
			175 - 0xAF
			187 - 0xBB: MERGEPAINT
			204 - 0xCC: SRCCOPY
			221 - 0xDD
			238 - 0xEE: SRCPAINT
			240 - 0xF0: PATCOPY
			245 - 0xF5

250 - 0xFA |
| | | | 255 - 0xFF: WHITENESS |

CB:CB_COLOR_CONTROL 寄存器用于控制颜色缓冲的行为，包括颜色处理模式、混合操作和其他相关设置。可以根据应用程序的需要配置这些字段，以影响渲染的结果。

CB_COLOR[0-7]_ATTRIB

CB:CB_COLOR[0-7]_ATTRIB 是一组可读写的 32 位寄存器，用于配置颜色缓冲区（COLOR surface）和相关属性的信息，其中 RT0 对应 CB:CB_COLOR0_ATTRIB，RT1 对应 CB:CB_COLOR1_ATTRIB，以此类推。这些寄存器的地址范围从 0x28c74 到 0x28e18。

以下是字段的定义：

字段名称	位范围	默认值	描述
TILE_MODE_INDEX	4:0	none	用于查找 GB_TILE_MODEn 寄存器以配置 COLOR 和 CMASK surface 的平铺设置的索引。
FMASK_TILE_MODE_INDEX	9:5	none	用于查找 GB_TILE_MODEn 寄存器以配置 FMASK surface 的平铺设置的索引。
NUM_SAMPLES	14:12	none	指定样本数量的对数。这个值不能大于 4（即 16 个样本）。
NUM_FRAGMENTS	16:15	none	指定片段数量的对数。这个值不能大于 MIN(NUM_SAMPLES, 3)，因为 log2(3) == 8 片段。
FORCE_DST_ALPHA_1	17	none	如果设置了此标志，将强制 DST_ALPHA=1.0f。适用于不具有 alpha 分量的格式。

CB_COLOR[0-7]_ATTRIB 寄存器用于配置与颜色缓冲区相关的属性，包括平铺模式、样本数量和片段数量等。这些属性的配置可以影响渲染的质量和性能。通过调整这些属性，可以满足不同渲染需求的要求。

CB_COLOR[0-7]_BASE

CB:CB_COLOR[0-7]_BASE 是一组可读写的 32 位寄存器，用于配置颜色缓冲区（COLOR surface）的基地址，其中 RT0 对应 CB:CB_COLOR0_BASE，RT1 对应 CB:CB_COLOR1_BASE，以此类推。这些寄存器的地址范围从 0x28c60 到 0x28e04。

以下是字段的定义：

字段名称	位范围	默认值	描述
BASE_256B	31:0	none	该字段的 8 位到 39 位（共 32 位）指定了设备地址空间中 COLOR surface 的起始字节地址的高位部分。
PIPE_SWZ	p-1:0		如果使用了管道（pipe） swizzle，则这些位（p-1:0）指定了管道 swizzle。p 的值等于 log2(numPipes)，其中 numPipes 是管道数量。
BANK_SWZ	p+b-1:p		如果使用了银行（bank） swizzle，则这些位（p+b-1:p）指定了银行 swizzle。b 的值等于 log2(numBanks)，其中 numBanks 是银行数量。

BASE_256B 寄存器的主要作用是配置 COLOR surface 的起始地址，以指示 GPU 在设备内存中查找渲染目标的颜色数据。这些寄存器的配置可以影响渲染目标的读写行为，并可以使用 swizzle 选项进行微调，以便更好地优化内存访问。

CB_COLOR[0-7]_CLEAR_WORD0

CB:CB_COLOR[0-7]_CLEAR_WORD0 是一组可读写的 32 位寄存器，用于配置清除颜色的低位（Bits [31:0]）数据。每个渲染目标都有对应的 CLEAR_WORD0 寄存器，其中 RT0 对应 CB:CB_COLOR0_CLEAR_WORD0，RT1 对应 CB:CB_COLOR1_CLEAR_WORD0，以此类推。这些寄存器的地址范围从 0x28c8c 到 0x28e30。

以下是字段的定义：

字段名称	位范围	默认值	描述
CLEAR_WORD0	31:0	none	清除颜色的低位数据（Bits [31:0]）。

CLEAR_WORD0 寄存器的作用是配置清除颜色的低位数据，这些数据用于在执行清除操作时覆盖渲染目标的颜色数据。这些寄存器的配置可以影响渲染目标的清除行为。

CB_COLOR[0-7]_CLEAR_WORD1

CB:CB_COLOR[0-7]_CLEAR_WORD1 是一组可读写的 32 位寄存器，用于配置清除颜色的高位（Bits [63:32]）数据。每个渲染目标都有对应的 CLEAR_WORD1 寄存器，其中 RT0 对应 CB:CB_COLOR0_CLEAR_WORD1，RT1 对应 CB:CB_COLOR1_CLEAR_WORD1，以此类推。这些寄存器的地址范围从 0x28c90 到 0x28e34。

以下是字段的定义：

字段名称	位范围	默认值	描述
CLEAR_WORD1	31:0	none	清除颜色的高位数据（Bits [63:32]）。

CLEAR_WORD1 寄存器的作用是配置清除颜色的高位数据，这些数据用于在执行清除操作时覆盖渲染目标的颜色数据。这些寄存器的配置可以影响渲染目标的清除行为。

CB_COLOR[0-7]_CMASK

CB:CB_COLOR[0-7]_CMASK 寄存器是一组可读写的 32 位寄存器，用于配置 CMASK（Color Mask）表面的基地址，其中 RT0 对应 CB:CB_COLOR0_CMASK，RT1 对应 CB:CB_COLOR1_CMASK，以此类推。这些寄存器的地址范围从 0x28c7c 到 0x28e20。

以下是字段的定义：

字段名称	位范围	默认值	描述
BASE_256B	31:0	none	指定设备地址空间中每个瓦片 CMASK 数据的起始字节地址的位 [39:8]。

CB:CB_COLOR[0-7]_CMASK 寄存器用于配置 CMASK 表面的基地址，CMASK 是一个用于渲染中的颜色蒙版数据的缓冲区。这些寄存器允许您定义每个渲染目标的 CMASK 表面的基地址，以便在渲染过程中使用 CMASK 数据来控制像素的写入。

CB_COLOR[0-7]_CMASK_SLICE

CB:CB_COLOR[0-7]_CMASK_SLICE 寄存器是一组可读写的 32 位寄存器，用于配置 CMASK（Color Mask）表面的切片大小，其中 RT0 对应 CB:CB_COLOR0_CMASK_SLICE，RT1 对应 CB:CB_COLOR1_CMASK_SLICE，以此类推。这些寄存器的地址范围从 0x28c80 到 0x28e24。

以下是字段的定义：

字段名称	位范围	默认值	描述
TILE_MAX	13:0	none	编码了切片的大小。该字段等于每个切片的 CMASK 数据的 128x128 块（16x16 块）的数量减 1。

CB:CB_COLOR[0-7]_CMASK_SLICE 寄存器用于配置 CMASK 表面的切片大小。CMASK 是一个用于渲染中的颜色蒙版数据的缓冲区。这些寄存器允许您定义每个渲染目标的 CMASK 表面的切片大小，以满足特定的渲染需求。

CB_COLOR[0-7]_FMASK

CB:CB_COLOR[0-7]_FMASK 寄存器是一组可读写的 32 位寄存器，用于配置 FMASK（Fragment Mask）表面的基地址，其中 RT0 对应 CB:CB_COLOR0_FMASK，RT1 对应 CB:CB_COLOR1_FMASK，以此类推。这些寄存器的地址范围从 0x28c84 到 0x28e28。

以下是字段的定义：

字段名称	位范围	默认值	描述
BASE_256B	31:0	none	这指定了资源在设备地址空间中起始位置的字节地址的[39:8]位。您可以在此处指定管道和银行的分布。具体来说：
			- 该字段的低位 [p-1:0]，其中 p = log2(numPipes)，指定了管道分布。
			- 该字段的 [p+b-1:p] 位，其中 b = log2(numBanks)，指定了银行分布。

CB:CB_COLOR[0-7]_FMASK 寄存器用于配置 FMASK 表面的基地址。FMASK 是一个用于遮挡和光栅化的辅助缓冲区，通常用于多重采样渲染中。这些寄存器允许您定义每个渲染目标的 FMASK 表面的基地址，以满足特定的渲染需求。其中包括管道和银行的分布设置。

CB_COLOR[0-7]_FMASK_SLICE

CB:CB_COLOR[0-7]_FMASK_SLICE 寄存器是一组可读写的 32 位寄存器，用于配置 FMASK（Fragment Mask）表面切片的大小。类似地，对于每个渲染目标，例如 CB:CB_COLOR0_FMASK_SLICE 对应 RT0，CB:CB_COLOR1_FMASK_SLICE 对应 RT1，以此类推。这些寄存器的地址范围从 0x28c88 到 0x28e2c。

以下是字段的定义：

字段名称	位范围	默认值	描述
TILE_MAX	21:0	none	编码了切片的大小。该字段等于每个切片的 8x8 瓦片的数量减一。

CB:CB_COLOR[0-7]_FMASK_SLICE 寄存器用于配置 FMASK 表面切片的大小。FMASK 是一个用于遮挡和光栅化的辅助缓冲区，通常用于多重采样渲染中。这些寄存器允许您定义每个渲染目标的 FMASK 表面切片的大小，以满足特定的渲染需求。

CB_COLOR*[0-7]*INFO

CB:CB_COLOR[0-7]_INFO 寄存器用于描述颜色渲染目标 (RT) 0 的表面格式信息，类似地，RT1-7 也有相似的寄存器。这些寄存器的地址范围从 0x28c70 到 0x28e14。

以下是字段的定义：

字段名称	位范围	默认值	描述
ENDIAN	1:0	none	指定在不同大小端模式下是否执行字节交换，字节交换相当于计算 dest[A] = src[A XOR N]，其中 A 是字节地址，N 是下面列出的 XOR 值。
			可能的值：
			00 - ENDIAN_NONE: 无字节交换 (XOR 0)
			01 - ENDIAN_8IN16: 在 16 位字内部进行 8 位交换 (XOR 1): 0xAABBCCDD -> 0xBBAADDCC
			02 - ENDIAN_8IN32: 在 32 位字内部进行 8 位交换 (XOR 3): 0xAABBCCDD -> 0xDDCCBBAA
			03 - ENDIAN_8IN64: 在 64 位字内部进行 8 位交换 (XOR 7): 0xaabbccddeeffgghh -> 0xhhggffeeddccbbaa
FORMAT	6:2	none	指定颜色分量的大小，在某些情况下还指定数字格式。请参阅下面的 COMP_SWAP 字段，了解将 RGBA（XYZW）着色器管线结果映射到像素格式中的颜色分量位置。从表面读取时，格式中缺失的分量将被默认值替代：RGB 为 0.0 或 Alpha 为 1.0。
			可能的值：
			00 - COLOR_INVALID: 此资源已禁用
			01 - COLOR_8: 标准化、整数
			02 - COLOR_16: 标准化、整数、浮点
			03 - COLOR_8_8: 标准化、整数
			04 - COLOR_32: 整数、浮点
			05 - COLOR_16_16: 标准化、整数、浮点

字段名称	位范围	默认值	描述
			06 - COLOR_10_11_11: 仅浮点
			07 - COLOR_11_11_10: 仅浮点
			08 - COLOR_10_10_10_2: 标准化、整数
			09 - COLOR_2_10_10_10: 标准化、整数
			10 - COLOR_8_8_8_8: 标准化、整数、sRGB
			11 - COLOR_32_32: 整数、浮点
			12 - COLOR_16_16_16_16: 标准化、整数、浮点
			13 - RESERVED
			14 - COLOR_32_32_32_32: 整数、浮点
			15 - RESERVED
			16 - COLOR_5_6_5: 仅标准化
			17 - COLOR_1_5_5_5: 仅标准化，1 位组件始终是 unorm
			18 - COLOR_5_5_5_1: 仅标准化，1 位组件始终是 unorm

19 - COLOR_4_4_4_4: 仅标准化 |
| | | | 20 - COLOR_8_24: unorm 深度，uint 模板
|
| | | | 21 - COLOR_24_8: unorm 深度，uint 模板
|
| | | | 22 - COLOR_X24_8_32_FLOAT: 浮点深度，uint 模板
|
| | | | 23 - RESERVED |
| LINEAR_GENERAL | 7 | none | 1: 覆盖 ARRAY_MODE 为 ARRAY_LINEAR_GENERAL，忽略 CB_COLOR0_ATTRIB.TILE_MODE_INDEX 设置。 |
| | | | 0: ARRAY_MODE=GB_TILE_MODE[CB_COLOR0_ATTRIB.TILE_MODE_INDEX].ARRAY_MODE
|
| NUMBER_TYPE | 10:8 | none | 指定颜色分量的数字类型。
|
			可能的值：
			00 - NUMBER_UNORM: 无符号重复小数 (urf)：范围 [0..1]，尺度因子 (2^n)-1
			01 - NUMBER_SNORM: Microsoft 风格的有符号小数 (rf)：范围 [-1..1]，尺度因子 (2^(n-1))-1
			02 - 保留
			03 - 保留
			04 - NUMBER_UINT: 零扩展位域，着色器中为 int：不可混合或可过滤

|
| | | | 05 - NUMBER_SINT: 符号扩展位域，着色器中为 int：不可混合或可过滤
|
			06 - NUMBER_SRGB: 伽马校正，范围 [0..1]（仅支持 COLOR_8_8_8_8 格式；总是将颜色通道四舍五入）
			07 - NUMBER_FLOAT: 浮点数：32 位：IEEE 浮点，SE8M23，偏置 127，范围 (-2^129..2^129)；16 位：Short 浮点 SE5M10，偏置 15，范围 (-2^17..2^17)；11 位：打包浮点，E5M6 偏置 15，范围 [0..2^17)；10 位：打包浮点，E5M5 偏置 15，范围 [0..2^17)
COMP_SWAP	12:11	none	指定如何将着色器中的红、绿、蓝和 Alpha 分量映射到渲染目标像素格式中的组件位置（0、1、2、3，其中 0 最不显著，3 最显著）。
			使用一个分量时，这将选择映射到单个渲染目标分量 (STD: R=>0; ALT: G=>0; STD_REV: B=>0; ALT_REV: A=>0)。 对于 2-4 个分量，SWAP_STD 始终将着色器分量映射到从 R=>0 开始的可用分量数量（分量 R=>0、G=>1、B=>2、A=>3）。 对于 2-3 个分量，SWAP_ALT 与 SWAP_STD 类似，除了来自着色器的 Alpha 总是发送到最后的渲染目标分量（2 个分量：R=>0、A=>1；3 个分量：R=>0、G=>1、A=>2）。 对于 4 个分量，SWAP_ALT 选择一种替代顺序（B=>0、G=>1、R=>2、A=>3）。 对于 2-4 个分量，SWAP_STD_REV 和 SWAP_ALT_REV 反转分量

顺序。 |
| | | | 可能的值： |
| | | | 00 - SWAP_STD: 标准小端组件顺序
|
| | | | 01 - SWAP_ALT: 替代组件或顺序
|
| | | | 02 - SWAP_STD_REV: 反转 SWAP_STD 顺序
|
| | | | 03 - SWAP_ALT_REV: 反转 SWAP_ALT 顺序
|
| FAST_CLEAR | 13 | none | 启用快速清除。如果设置，CB 识别 cmask 中的快速清除编码，并将相应的瓦片区域视为已快速清除。 |
| COMPRESSION | 14 | none | 启用颜色压缩。
|
BLEND_CLAMP	15	none	指定是否在混合之前将源数据夹紧到格式范围内，以及混合后夹紧。此位必须在 BLEND_BYPASS 设置为 true 时清除。否则，它必须在任何分量为 SINT/UINT（NUMBER_TYPE = SINT、UINT，或 FORMAT = COLOR_8_24、COLOR_24_8、COLOR_X24_8_32_FLOAT） 时设置。
BLEND_BYPASS	16	none	如果为 false，则该 MRT 的混合器根据 CB_BLENDn_CONTROL.ENABLE 中的指定启用/禁用。如果为 true，则禁用混合。当且仅当任何分量为 SINT/UINT（NUMBER_TYPE = SINT、UINT，或 FORMAT = COLOR_8_24、COLOR_24_8、COLOR_X24_8_32_FLOAT） 时，应设置此位。
SIMPLE_FLOAT	17	0x0	如果设置，通过忽略特殊值（如 NaN、+/-Inf 和 -0.0f）来简化浮点处理，以使 DESTBLEND*DST=0.0f 如果 DESTBLEND==0.0f 以及 SRCBLEND*SRC=0.0f 如果 SRCBLEND==0.0f。如果为 false，则浮点处理遵循特殊值（如 NaN、+/-Inf 和 -0.0f）的完整 IEEE 规则。对于浮点表面，设置此字段可以帮助启用以下混合优化：BLEND_OPT_DONT_RD_DST、BLEND_OPT_BYPASS 和 BLEND_OPT_DISCARD_PIXEL。对于其他组件格式，将忽略此位。
ROUND_MODE	18	none	此字段选择将混合器结果转换为帧缓冲组件时的截断（浮点数的标准方式）和四舍五入之间的方式。如果任何分量为 UNORM、SNORM 或 SRGB，应将其设置为 ROUND_BY_HALF（此字段对于 COLOR_8_24 和 COLOR_24_8 被忽略）。
			可能的值：
			00 - ROUND_BY_HALF: 加上 1/2 最低有效位，然后截断

			01 - ROUND_TRUNCATE: 截断到浮点数零值
CMASK_IS_LINEAR	19	none	如果设置，Cmask 表面存储为线性存储。这可以减少 cmask 表面上的填充限制。
BLEND_OPT_DONT_RD_DST	22:20	0x0	不读取 DST 的混合优化：如果混合函数计算为 SRCBLEND*SRC +/- 0*DST 并且 SRBBLEND 也不需要 DST，那么不要读取 DST。
			可能的值：
			00 - FORCE_OPT_AUTO: （默认）硬件自动检测并启用此优化
			01 - FORCE_OPT_DISABLE: 禁用此 RT 的优化
			02 - FORCE_OPT_ENABLE_IF_SRC_A_0: 仅在 Src Alpha 为 0.0f 时启用优化
			03 - FORCE_OPT_ENABLE_IF_SRC_RGB_0: 仅在 Src 颜色分量 (RGB) 都为 0.0f 时启用优化
			04 - FORCE_OPT_ENABLE_IF_SRC_ARGB_0: 仅在 Src 颜色分量 (RGB) 和 Alpha 都为 0.0f 时启用优化
			05 - FORCE_OPT_ENABLE_IF_SRC_A_1: 仅在 Src Alpha 为 1.0f 时启用优化
			06 - FORCE_OPT_ENABLE_IF_SRC_RGB_1: 仅在 Src 颜色分量 (RGB) 都为 1.0f 时启用优化
			07 - FORCE_OPT_ENABLE_IF_SRC_ARGB_1: 仅在 Src 颜色分量 (RGB) 和 Alpha 都为 1.0f 时启用优化
BLEND_OPT_DISCARD_PIXEL	25:23	0x0	丢弃像素的混合优化：如果混合函数计算为 0*SRC +/- 1*DST，则这将成为 NOP。
			可能的值：
			00 - FORCE_OPT_AUTO: （默认）硬件自动检测并启用此优化
			01 - FORCE_OPT_DISABLE: 禁用此 RT 的优化
			02 - FORCE_OPT_ENABLE_IF_SRC_A_0: 仅在 Src Alpha 为 0.0f 时启用优化
			03 - FORCE_OPT_ENABLE_IF_SRC_RGB_0: 仅在 Src 颜色分量 (RGB) 都为 0.0f 时启用优化
			04 - FORCE_OPT_ENABLE_IF_SRC_ARGB_0: 仅在 Src 颜色分量 (RGB) 和 Alpha 都为 0.0f 时启用优化
			05 - FORCE_OPT_ENABLE_IF_SRC_A_1: 仅在 Src Alpha 为 1.0f 时启用优化
			06 - FORCE_OPT_ENABLE_IF_SRC_RGB_1: 仅在 Src 颜色分量 (RGB) 都为 1.0f 时启用优化
			07 - FORCE_OPT_ENABLE_IF_SRC_ARGB_1: 仅在 Src 颜色分量 (RGB) 和 Alpha 都为 1.0f 时启用优化

这些字段允许配置颜色渲染目标的各种特性，包括颜色格式、字节顺序、数字类型、混合和优化选项等。这些设置可根据渲染需求和硬件支持进行调整，以获得最佳性能和效果。

CB_COLOR[0-7]_PITCH

CB:CB_COLOR[0-7]_PITCH 寄存器是一组可读写的 32 位寄存器，用于控制渲染目标颜色缓冲区的扫描线间距（Pitch）。每个寄存器对应一个渲染目标，例如，CB:CB_COLOR0_PITCH 对应 RT0，CB:CB_COLOR1_PITCH 对应 RT1，以此类推。这些寄存器的地址范围从 0x28c64 到 0x28e08。

以下是字段的定义：

字段名称	位范围	默认值	描述
TILE_MAX	10:0	none	编码了扫描线的间距（Pitch）；如果 Pitch 是每个扫描线的数据元素数，那么这个字段是 (Pitch / 8) - 1，表示X维度上允许的最大 8x8 瓦片编号。

CB:CB_COLOR[0-7]_PITCH 寄存器用于配置渲染目标颜色缓冲区的扫描线间距，以便在渲染过程中正确处理像素数据的存储。不同的渲染目标可以具有不同的扫描线间距，以满足渲染需求。

CB_COLOR[0-7]_SLICE

CB:CB_COLOR[0-7]_SLICE 寄存器是一组可读写的 32 位寄存器，用于控制渲染目标颜色缓冲区的切片大小（Slice）。每个寄存器对应一个渲染目标，例如，CB:CB_COLOR0_SLICE 对应 RT0，CB:CB_COLOR1_SLICE 对应 RT1，以此类推。这些寄存器的地址范围从 0x28c68 到 0x28e0c。

以下是字段的定义：

字段名称	位范围	默认值	描述
TILE_MAX	21:0	none	编码了切片的大小。如果 SliceTiles 是一个切片中的最大瓦片数（等于 Pitch * Height / 64），那么这个字段是 SliceTiles - 1，表示允许的最大切片中的瓦片编号。

CB:CB_COLOR[0-7]_SLICE 寄存器用于配置渲染目标颜色缓冲区的切片大小，以便在渲染过程中正确管理渲染目标的数据存储。不同的渲染目标可以具有不同的切片大小，以满足渲染需求。

CB_COLOR[0-7]_VIEW

CB:CB_COLOR[0-7]_VIEW 寄存器是一组可读写的 32 位寄存器，用于选择渲染目标的切片索引范围。每个寄存器对应一个渲染目标，例如，CB:CB_COLOR0_VIEW 对应 RT0，CB:CB_COLOR1_VIEW 对应 RT1，以此类推。这些寄存器的地址范围从 0x28c6c 到 0x28e10。

以下是字段的定义：

字段名称	位范围	默认值	描述
SLICE_START	10:0	none	对于 ARRAY_LINEAR_GENERAL：此字段的[7:0]位指定资源的字节地址的[7:0]位。与 CB_COLOR*_BASE.BASE_256B 一起使用，指定 40 位起始地址。地址必须是元素对齐的。使用 ARRAY_LINEAR_GENERAL 时，由于 SLICE_START 没有实际值，因此在进行 rtindex（切片）夹紧时，假定 SLICE_START 值为零。对于所有其他表面，此字段指定了此视图的起始切片编号：此字段添加到 rtindex 以计算要呈现的切片。
SLICE_MAX	23:13	none	指定此资源的允许的最大渲染目标切片索引（rtindex），这比总切片数少一个。如果超过此值，rtindex 将被夹紧到 SLICE_START。

CB:CB_COLOR[0-7]_VIEW 寄存器用于选择要呈现的渲染目标的切片索引范围。这些寄存器允许配置渲染目标的子区域以实现高级渲染技术，例如多视图渲染或渲染到纹理数组的特定切片。

CB_TARGET_MASK

CB:CB_TARGET_MASK 寄存器是一个可读写的 32 位寄存器，用于控制写入多渲染目标 (MRTs) 的颜色组件掩码。该寄存器的地址为 0x28238。

以下是字段的定义：

字段名称	位范围	默认值	描述
TARGET0_ENABLE	3:0	none	启用对 RT 0 组件的写入。低位对应红色通道。0 位禁止对该通道的写入，1 位启用对该通道的写入。
TARGET1_ENABLE	7:4	none	启用对 RT 1 组件的写入。
TARGET2_ENABLE	11:8	none	启用对 RT 2 组件的写入。
TARGET3_ENABLE	15:12	none	启用对 RT 3 组件的写入。
TARGET4_ENABLE	19:16	none	启用对 RT 4 组件的写入。
TARGET5_ENABLE	23:20	none	启用对 RT 5 组件的写入。
TARGET6_ENABLE	27:24	none	启用对 RT 6 组件的写入。
TARGET7_ENABLE	31:28	none	启用对 RT 7 组件的写入。

CB:CB_TARGET_MASK 寄存器允许控制多渲染目标的颜色组件写入，通过配置每个组件的使能位来选择是否写入对应的渲染目标。这些掩码字段使您可以选择哪些颜色通道会被写入到每个渲染目标中。

DB_DEPTH_INFO

DB:DB_DEPTH_INFO 寄存器是一个32位寄存器，用于配置深度缓冲区信息，特别是与数据存储有关的配置。该寄存器的 GPU 寄存器地址为 0x2803c。

字段解析

Field Name	位范围	默认值	描述
ADDR5_SWIZZLE_MASK	3:0	none	对于32B瓦片，指示数据应存储在64B字的上半部分还是下半部分。如果ADDR5_SWIZZLE_MASK与{TILE_Y[1:0], TILE_X[1:0]}的异或减少设置，使用上半部分，否则使用下半部分。最可能的最佳值是0x1。

DB:DB_DEPTH_INFO 寄存器用于配置深度缓冲区数据的存储方式。特别地，ADDR5_SWIZZLE_MASK 字段用于指示在32B瓦片中数据应存储在64B字的哪一半，根据一些特定条件进行选择。这个寄存器的配置可以影响深度缓冲区数据的组织和存储方式。

DB_DEPTH_SIZE

DB:DB_DEPTH_SIZE 寄存器是一个32位寄存器，用于配置深度缓冲区的大小。该寄存器的 GPU 寄存器地址为 0x28058。

字段解析

Field Name	位范围	默认值	描述
PITCH_TILE_MAX	10:0	none	以8x8像素瓦片为单位的宽度。实际宽度为 (Pitch/8 - 1)。
HEIGHT_TILE_MAX	21:11	none	以8x8像素瓦片为单位的深度缓冲区高度。实际高度为 (height/8 - 1)。

DB:DB_DEPTH_SIZE 寄存器用于配置深度缓冲区的大小，特别是指定深度缓冲区的宽度和高度。PITCH_TILE_MAX 字段表示深度缓冲区的宽度（以8x8像素瓦片为单位），而 HEIGHT_TILE_MAX 字段表示深度缓冲区的高度（同样以8x8像素瓦片为单位）。这些字段的配置决定了深度缓冲区的存储尺寸。

DB_DEPTH_SLICE

DB:DB_DEPTH_SLICE 寄存器是一个32位寄存器，用于配置深度缓冲区切片的信息。该寄存器的 GPU 寄存器地址为 0x2805c。

字段解析

Field Name	位范围	默认值	描述
SLICE_TILE_MAX	21:0	none	到下一个切片的8x8像素瓦片数加上一些小数以能够旋转瓦片模式。 (pitch*height/64 - 1)。

DB:DB_DEPTH_SLICE 寄存器用于配置深度缓冲区切片的信息，特别是指定了在下一个切片之前有多少个8x8像素瓦片。这个值被计算为 (pitch*height/64 - 1) 加上一些小数，以便能够旋转瓦片模式。深度缓冲区切片的配置通常与深度缓冲区的大小和布局有关，影响图形渲染中深度信息的存储方式。

DB_DEPTH_VIEW

DB:DB_DEPTH_VIEW 寄存器是一个32位寄存器，用于选择渲染目标0的切片索引范围。该寄存器的 GPU 寄存器地址为 0x28008。

字段解析

Field Name	位范围	默认值	描述
SLICE_START	10:0	none	指定此视图的起始切片编号。此字段将添加到 RenderTargetArrayIndex 以计算要渲染的切片。SLICE_START 必须小于或等于 SLICE_MAX。
SLICE_MAX	23:13	none	指定此资源的最大允许的 Z 切片索引，即总切片数减一。
Z_READ_ONLY	24	none	只读 Z 缓冲区，即禁止写入 Z 缓冲区。
STENCIL_READ_ONLY	25	none	只读模板缓冲区，即禁止写入模板缓冲区。

DB:DB_DEPTH_VIEW 寄存器用于选择渲染目标0的切片索引范围，并可以配置是否只读的 Z 缓冲区和模板缓冲区。SLICE_START 和 SLICE_MAX 字段用于指定切片范围，Z_READ_ONLY 和 STENCIL_READ_ONLY 字段用于配置只读属性。这些配置影响深度和模板测试的行为。

DB_EQAA

DB:DB_EQAA 寄存器是一个32位寄存器，用于控制深度缓冲区的等效抗锯齿 (EQAA) 相关设置。EQAA 是一种用于提高图形渲染质量的技术，通过对多个样本进行采样和插值来实现更平滑的图像效果。该寄存器的 GPU 寄存器地址为 0x28804。

字段解析

Field Name	位范围	默认值	描述
MAX_ANCHOR_SAMPLES	2:0	none	设置CB允许使用的锚定样本数的最大值。将其设置为DB表面分配的最小样本数以限制CB在创建非锚定片段后可能需要丢弃它们。
PS_ITER_SAMPLES	6:4	none	指定PS_ITER_SAMPLE设置时要迭代的样本数量，从而设置超采样量。通常这是应用程序公开的样本数量。不支持大于深度表面样本数的值。
MASK_EXPORT_NUM_SAMPLES	10:8	none	指定用于着色器掩码导出的样本数量。
ALPHA_TO_MASK_NUM_SAMPLES	14:12	none	为A2M生成的质量样本数。将其设置在应用程序公开的样本数和较高EQAA样本数之间以进行速度/质量权衡。如果ALPHA_TO_MASK_EQAA_DISABLE=1，则必须将其设置为应用程序公开的样本数。

Field Name	位范围	默认值	描述
HIGH_QUALITY_INTERSECTIONS	16	none	如果未设置，所有完全覆盖的瓦片将以瓦片速率通过详细步进器运行，仅在深度测试结果未知时才以DB的表面速率减慢，或者在着色器执行时以像素速率减慢。如果设置，只会加速已知Z测试结果的完全覆盖瓦片，但仍允许可能存在Z交点的瓦片以详细速率运行，从而获得抗锯齿的交点。最好与INTERPOLATE_COMP_Z一起使用以获得最佳质量。
INCOHERENT_EQAA_READS	17	none	禁用对共享锚定样本但不共享详细样本的相邻三角形的一致性检查，对于不存在数据转发的相邻条带来说很重要。可能会引入依赖于延迟的结果，因此除了单元测试外，应强制为0。
INTERPOLATE_COMP_Z	18	none	允许未锚定的样本从压缩的Z平面插值出唯一的Z值。在像素的第一个交点上创建漂亮的抗锯齿交点。引入依赖于延迟的结果，因此除了单元定向测试可能需要可视检查外，应将其强制为0。
INTERPOLATE_SRC_Z	19	none	强制未锚定样本在目标Z未压缩时插值出唯一的源Z，以获得更平滑的交点，即使在未压缩的Z上也会引发ZFUNC==EQUALS的混合失败。除非进行实验，否则可能永远不会设置。
STATIC_ANCHOR_ASSOCIATIONS	20	none	强制复制的目标数据始终来自静态关联的锚定样本，而不是尝试从最靠近的锚定样本拉取目标数据，该样本位于原始图元内。当设置时，可能会导致额外的一致性停顿并且可能会降低相邻三角形的质量。
ALPHA_TO_MASK_EQAA_DISABLE	21	none	使Alpha to Mask设置的样本与先前的GPU完全相同。只有在需要先前一代的行为时才应设置，否则新行为针对EQAA进行了优化，可以在混合AA模式和即使在没有AA时也可以提高质量。
OVERRASTERIZATION_AMOUNT	26:24	none	超采样中的样本掩码的OR减少次数的对数。
ENABLE_POSTZ_OVERRASTERIZATION	27	none	启用PostZ中的超采样（即，在着色器之后）。

DB:DB_EQAA 寄存器用于控制深度缓冲区的等效抗锯齿 (EQAA) 相关设置。EQAA 技术可以提高图形渲染的质量，通过对多个样本进行采样和插值来实现更平滑的图像效果。这些字段允许配置 EQAA 的不同方面，以满足特定的渲染需求和质量要求。

DB_Z_INFO

DB:DB_Z_INFO 寄存器是一个可读写的 32 位寄存器，用于控制深度缓冲区（Z 缓冲区）的属性和行为。该寄存器的地址为 0x28040。

以下是字段的定义：

字段名称	位范围	默认值	描述
FORMAT	1:0	none	指定深度分量的大小以及深度是否为浮点数。
			可能的值：
			00 - Z_INVALID：无效的深度表面。

字段名称	位范围	默认值	描述
			01 - Z_16：16 位 UNORM 深度表面。
			02 - Z_24：（已弃用：请改用 Z_32_FLOAT）24 位 UNORM 深度表面。
			03 - Z_32_FLOAT：32 位浮点深度表面。
NUM_SAMPLES	3:2	none	指定 Z 表面的多重采样（MSAA）表面占用。
TILE_MODE_INDEX	22:20	none	指示此表面将使用的 GB_TILE_MODEn 寄存器的索引。
ALLOW_EXPCLEAR	27	none	允许 ZMask 跟踪扩展和清除。
READ_SIZE	28	none	设置读取的最小大小为 512 位。如果该表面位于具有小于 512 位访问的内存池中，则设置此选项。
			可能的值：
			00 - READ_256_BITS
			01 - READ_512_BITS
TILE_SURFACE_ENABLE	29	none	启用读写 htile 数据。如果关闭，HiZ+S 也将关闭。
ZRANGE_PRECISION	31	none	0 = ZMin 是基础值，通常在执行 Z > 测试时设置。
			1 = ZMax 是基础值，通常在执行 Z < 测试时设置。
			基础值具有完整的 14 位精度。通过将基础值设置为 Max，在 < 测试中剔除时误差较小。
			仅可在完全表面清除后更改此字段。如果 TILE_Z_ONLY == 0，则此字段才具有意义。

DB:DB_Z_INFO 寄存器允许您配置深度缓冲区的格式、多重采样属性以及其他深度缓冲区相关的选项。这些设置可以影响深度测试和渲染的行为。

DB_STENCIL_INFO

DB:DB_STENCIL_INFO 寄存器是一个可读写的 32 位寄存器，用于配置模板测试过程中的模板缓冲区的属性。该寄存器的地址为 0x28044。

以下是字段的定义：

字段名称	位范围	默认值	描述
FORMAT	0	none	指定模板组件的大小。
			可能的值：
			00 - STENCIL_INVALID: 无效的模板表面。
			01 - STENCIL_8: 8 位整数模板表面。
TILE_MODE_INDEX	22:20	none	该表面将用于 tile_split 的 GB_TILE_MODEn 寄存器的索引。所有其他字段将来自 DB_Z_INFO 寄存器。
ALLOW_EXPCLEAR	27	none	允许模板内存格式跟踪扩展和清除。
TILE_STENCIL_DISABLE	29	none	表示 htile 缓冲区没有模板元数据。这提高了 hiz 的精度，但取消了模板压缩或 HiStencil 优化的使用。

DB_STENCIL_READ_BASE

DB:DB_STENCIL_READ_BASE 寄存器是一个可读写的 32 位寄存器，用于配置模板读取过程中的模板缓冲区的地址。该寄存器的地址为 0x2804c。

以下是字段的定义：

字段名称	位范围	默认值	描述
BASE_256B	31:0	none	模板读取操作中的模板表面的首字节位置，必须是 256 字节对齐的。40 位地址的高 32 位。

这个字段允许指定用于读取模板的模板缓冲区的起始地址，以便在渲染过程中读取模板数据。

DB_STENCIL_WRITE_BASE

寄存器信息	
寄存器名称	DB:DB_STENCIL_WRITE_BASE
位数	32 bits
访问权限	读/写（R/W）
访问方式	32
寄存器类别	GpuF0MMReg
地址	0x28054

字段名称	位	默认值	描述
BASE_256B	31:0	无	在设备地址空间中进行写入时Stencil表面的第一个字节的位置，必须256字节对齐。40位地址的高32位。

DB_Z_READ_BASE

寄存器信息	
寄存器名称	DB:DB_Z_READ_BASE
位数	32 bits
访问权限	读/写（R/W）
访问方式	32
寄存器类别	GpuF0MMReg
地址	0x28048

字段名称	位	默认值	描述
BASE_256B	31:0	无	在设备地址空间中进行读取时Z表面的第一个字节的位置，必须256字节对齐。40位地址的高32位。

DB_Z_WRITE_BASE

寄存器信息	
寄存器名称	DB:DB_Z_WRITE_BASE
位数	32 bits
访问权限	读/写（R/W）
访问方式	32
寄存器类别	GpuF0MMReg
地址	0x28050

字段名称	位	默认值	描述
BASE_256B	31:0	无	在设备地址空间中进行写入时Z表面的第一个字节的位置，必须256字节对齐。40位地址的高32位。