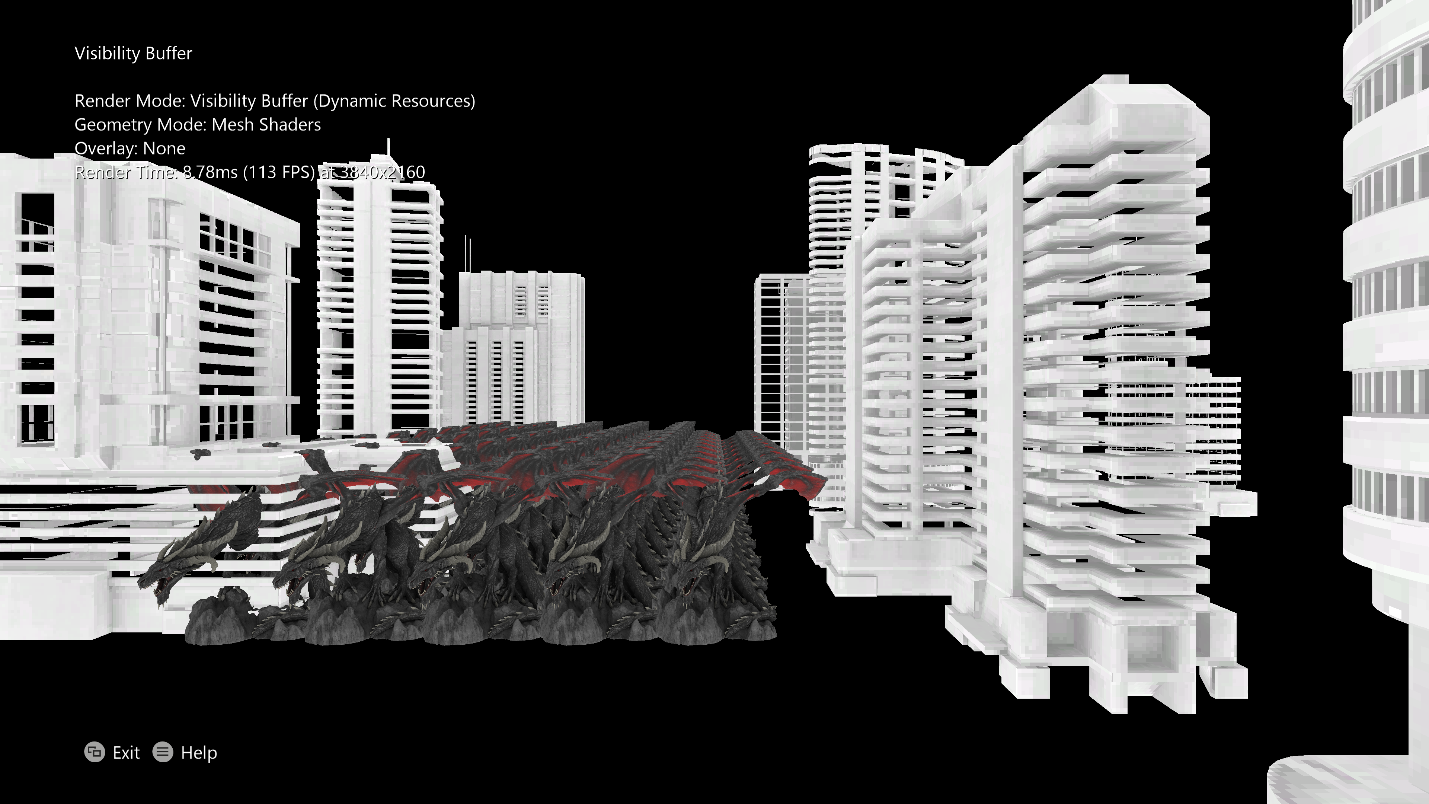
可见性缓冲区示例

*此示例与 Microsoft 游戏开发工具包（2021 年 4 月）兼容*

# 说明

此示例演示使用网格着色器和 HLSL 6.6 动态资源的可见性缓冲区 (延迟) 呈现技术。



# 生成示例

如果使用 Xbox One 开发工具包，请将活动解决方案平台设置为 Gaming.Xbox.XboxOne.x64。

如果使用 Xbox Series X|S，请将活动解决方案平台设置为 Gaming.Xbox.Scarlett.x64。

由于使用 HLSL SM 6.6 功能，为电脑构建 (Gaming.Desktop.x64) 需要 [DirectX Agility SDK](https://devblogs.microsoft.com/directx/gettingstarted-dx12agility/)。Direct Agility SDK 以 NuGet 程序包的形式包含在示例中。它还利用 Microsoft.Windows.SDK.CPP NuGet 程序包获取 DXC.exe 编译器的最新 Windows SDK (22000) 版本。开发人员还可以直接从 [Github](https://github.com/microsoft/DirectXShaderCompiler/releases) 使用最新的 DXC。

*有关详细信息，请参阅 GDK 文档中的*“运行示例”*。*

# 使用示例

该示例采用飞行/第一人称相机，以允许将相机放置在场景中的任意位置。下面列出了完整的控制说明：

# 控制

|  |  |
| --- | --- |
| 操作 | 游戏板 |
| 查看控制 | “开始”按钮 |
| 在可见性缓冲区（延迟）渲染和转发渲染之间切换。可见性缓冲区渲染使用动态资源。转发渲染则不使用。 | A |
| 循环可见性缓冲区覆盖。将基元 ID 或对象 ID 显示为覆盖。在内部，两者都渲染为单个 32 位 UINT 渲染目标。 | B |
| 切换顶点着色器/网格着色器几何图形渲染。仅适用于可见性缓冲区模式。仅 Xbox One 支持顶点着色器。 | X |
| 相机向前/向后/侧身 | 左摇杆 |
| 相机绕 Y 轴旋转/绕 X 轴旋转 | 右摇杆 |
| 相机向上/向下 | 左/右扳机键 |
| 快速飞行模式 (Turbo) | 单击 + 按住左摇杆 |
| 退出 | 视图按钮 |

# 实施说明

显示两个渲染路径（按 A 进行切换）。

在转发路径中，执行典型的光栅化，所有资源（索引缓冲区、顶点缓冲区、纹理、取样器等）都通过根签名或输入汇编程序绑定。

在可见性缓冲区路径中，执行“延迟”光栅化，并选择网格着色器或用于几何着色的顶点着色器，以及执行最终场景着色的计算着色器。如果使用网格着色器，则通过 HLSL 6.6 功能 ResourceDescriptorHeap[] 和 SamplerDescriptorHeap[]“无绑定地”访问所有资源。在顶点着色器模式下，由于性能原因，索引缓冲区和顶点缓冲区绑定到输入汇编程序。

在顶点着色器模式下，也可以只绑定索引缓冲区，并通过 ResourceDescriptorHeap[] 访问顶点缓冲区，但这样会导致某些硬件的性能严重损失。

Xbox One 不支持网格着色器，因此只提供顶点着色器模式。

可见性缓冲区

可见性缓冲区路径是“延迟”渲染器的实现。但是，初始光栅化路径仅输出单个 32 位 UINT 渲染目标（可见性缓冲区），其中包含前 12 位（4096 个可能的对象）中的一个对象标识符，以及最后 20 位（每个对象 1,048,576 个三角形）中的一个基元标识符。然后，计算着色器传递使用可见性缓冲区，从 ResourceDescriptorHeap 加载每个对象的常量信息，然后从 ResourceDescriptorHeap 加载索引缓冲区/顶点缓冲区/纹理。然后，计算着色器重新构造每个三角形的命中点，并确定适当的内插纹理坐标和屏幕空间导数。请注意，这与普通延迟渲染器不同，在第一次传递后，此数据将存储在中间渲染目标中。最后，使用 SamplerDescriptorHeap 加载 SamplerState 对象，并对纹理进行采样并输出到屏幕。

几何着色

此示例同时实现顶点着色器和网格着色器（如果支持）。可见性缓冲区必须包含基元 ID，该 ID 可以在顶点着色器传递后从 SV\_PrimitiveID 系统语义中获取，也可以作为网格着色器的每个基元输出传递。

对于网格着色器，执行的是简单的启动时间 Meshlet 拆分算法。此算法需要预先优化的索引缓冲区（例如使用 -op 标志的 DirectXMeshs 的 MeshConvert 的输出）。然后，会将索引缓冲区拆分成 Meshlet 以满足最大顶点/最大基元限制，同时保持现有的基元排序。然后，非常简单的网格着色器可以通过 ResourceDescriptorHeap[] 加载 Meshlet 信息，执行正常的顶点着色，并在每个基元导出中输出 primitiveID。此 ID 与原始索引缓冲区保持相同的排序，因此可以与相同的计算着色器一起使用，计算着色器无需了解 Meshlet 格式。

性能

此技术的性能取决于许多因素。网格着色器和顶点着色器之间的几何着色成本和权衡将取决于目标 GPU 平台和资产大小。

此外，转发渲染器的像素着色器的工作内容可以与顶点着色器的工作内容并行运行，而可见性缓冲区渲染器则必须在处理完所有顶点后，在计算中执行其“像素着色”。这意味着，除非像素工作量超过顶点工作量。否则，更大的顶点着色器工作量将会隐藏任何过度绘制成本。

但是，计算着色器非常高效，如果像素着色器的工作量占据主导地位，可见性缓冲区渲染器可以发挥更强的性能。这主要是由于完全缺乏过度绘制和网格着色器的细微性能优势造成的，但是也有助于避免四色帮助程序通道，以及可能会减慢像素着色器波次的基元输出的怪异。

在此示例中，转发渲染器和可见性缓冲区渲染器在其着色结束时的循环中都执行了多次冗余纹理提取。目的是为了增加像素着色器成本，以更好地反映过度绘制的性能差异，因为如果没有这种差异，顶点着色器的工作始终占据着色器的主导地位。

最后，尽管此示例中未实现传统的延期渲染器，但此方法可能也具有很多优势。关键注释的渲染目标内存要求要低得多，可见性缓冲区仅为单个 32 位目标，而不是用于漫射/普通/反射/等效果的多个目标。同样，所需的填充率也要低得多，因为在初始传递中只写入一个目标，而计算着色器只输出一个 UAV。必要时，需要直接从描述符堆中提取所需的资源。

# 隐私声明

在编译和运行示例时，将向 Microsoft 发送示例可执行文件的文件名以帮助跟踪示例使用情况。若要选择退出此数据收集，你可以删除 Main.cpp 中标记为“示例使用遥测”的代码块。

有关 Microsoft 的一般隐私策略的详细信息，请参阅 [Microsoft 隐私声明](https://privacy.microsoft.com/en-us/privacystatement/)。