Overview

In 3D rendering the term **culling** describes the early rejection of objects of any kind (objects, draw calls, triangles and pixels) that don't contribute to the final image. Many techniques exist that reject objects at various stages in the rendering pipeline. Some of those techniques are fully done in software on the CPU, some are hardware (GPU) assisted and some are built into the graphics card. It is helpful to understand all of those techniques in order to achieve good performance. To remove as much processing work as possible, it is better to cull early and to cull more. On the other hand the culling itself should not cost too much performance and memory. To ensure good performance, we automatically balance the system. This gives better performance but also makes the system characteristics a bit harder to understand.

Usually engine culling happens at the draw call level or on multiple draw calls. We don't go down to the triangle level or even further as this is often not efficient. That means it might make sense to **split huge objects** into multiple parts so that not all parts need to be rendered together.

Frustum Culling

Frustum culling is a basic technique that every serious 3D engine is doing. In its simplest form, all objects of the scene are tested for intersection with the view frustum pyramid. This test can be conservative which means that it is acceptable to have some objects being reported visible even if they are not. This allows optimizations like approximating the objects by more coarse bounding primitives. We use AABB's (axis aligned bounding boxes) and OBB's (oriented bounding boxes) when we test against the frustum pyramid. The test is done hierarchically which means first some coarse enclosing objects (e.g. a terrain sector) are tested for visibility while the testing is recursively refined down to the draw call level. Frustum culling is implemented in the 3D engine as it is hardware independent and the relevant structures (e.g. bounding box extents) are stored there.

No Static Culling Techniques

We deliberately avoid static techniques like precomputed PVS (Potential Visibility Set) that have been a common approach in early 3D engines. The major advantage of the PVS is the **very small run-time performance** cost but with modern computers this is less of a concern. Creating the PVS is usually done in a **time consuming pre-processing step** and that is bad for production in modern big scale game worlds. Gameplay often requires dynamic updates of the geometry (e.g. opening/closing doors, destroying buildings) and the **static nature** of a PVS is not very well suited for that. By avoiding using PVS and similar pre-computation based techniques, we can even **save some memory** which is quite valuable on consoles.

Portals

Another popular approach besides PVS are portals. Portals are usually hand-placed flat convex 2d objects. The idea is that if a portal is visible (simple to test as the object is quite primitive), the world section behind the portal needs to be rendered. If the world section is considered visible, the geometry can be tested further as the portal shape or the intersection of multiple portals allow for more culling. By separating world sections with portals, **designers can create efficient levels**. Good portal positions are in narrow areas like doors and in narrow walkways. The test against the portals is done recursively from front to back, which is why it is beneficial to create environments where portals cull away a lot of content. There are algorithms to automatically place portals but it is likely that the resulting level is less optimal in performance if the designers are not involved in the optimization process.

Hand placed portals **avoid time consuming pre-processing steps**, use **little extra memory** and allow some dynamic updates. Portals can be switched on and off (good for doors) and some engines even implement special effects with portals (e.g. mirrors, teleporters). In CryENGINE portals are exclusively used to improve rendering performance. We decided to not extend the use of portals any further in order to keep the code and portal workflow simple and efficient. Portals have their advantages but additional effort from designers is required and **often it is hard to find good portal positions**. Open environments like cities or pure nature often don't allow efficient portal usage. Portals are supported by CryENGINE but should only be used when they perform better than the coverage buffer (see below).

Anti-Portals

The portal technique can be extended by the opposite of portals which are commonly named anti-portals. Those objects which are often convex in 2d or 3d can occlude other portals. Imagine a big pillar in a room with multiple doors connecting to other rooms. This is a hard case for classic portals but the typical use case for anti-portals. Anti-portals can be implemented with geometric intersections of objects but that method has problems with the fusion of multiple anti-portals and efficiency suffers. Instead of geometric anti-portals, we have the coverage buffer which serves the same purpose but has superior characteristics.

GPU Z Test

In modern graphics cards the Z buffer is used to solve the hidden surface problem. Here a simple explanation: For each pixel on the screen the so called z or depth value is stored which represents the distance of the camera to the nearest geometry at that pixel location. All renderable objects need to be made out of triangles. All pixels covered by the triangles perform a z comparison (z buffer value vs. z value of the triangle) and depending on the result, the triangle pixel is rejected or not. This solves hidden surface removal elegantly even for intersecting objects. The earlier mentioned problem of occluder fusion is handled without further effort. The Z Test is quite late in the rendering pipeline which means a lot of engine setup (e.g. skinning, shader constants) cost is already done.

In some cases, it allows to avoid pixel shader execution or frame buffer blending but its **main purpose is to solve the hidden surface problem**, the culling is a side benefit. By sorting objects roughly from front to back, the culling performance can be improved. The early z pass (sometimes named z pre pass) technique makes this less indispensable as the first pass is explicitly fast on the per pixel performance. Some hardware even runs in double speed if color writes are disabled. Unfortunately we need to output data there to setup the G-Buffer data for the deferred lighting.

The z buffer **precision** is affected by the pixel format (24bit), the z buffer range and in a **very extreme (non linear) way** by the z near value. The z near value defines how close some object can be to the viewer before it gets clipped away. By halving the z near (e.g. from 10cm top 5cm), you effectively halve the precision of the z buffer. This won't affect most object rendering but decals are often only rendered correctly because their z value is slightly smaller than the surface below them. It is a good advise to not change the z near at run-time.
GPU Z Cull / HiZ

Efficient z buffer implementations in the GPU cull fragments (pixels or multi-sampled sub-samples) in more coarse blocks at an earlier stage. This helps to reduce pixel shader execution. There are many conditions required in order to allow this optimization and seemingly harmless renderer changes can easily break this optimization. The rules are complicated and depend on the graphics hardware.

GPU Occlusion Queries

The occlusion query feature of modern graphics cards allows the CPU to get back information on z buffer tests that have been done before. This feature can be used to implement more advanced culling techniques. After rendering some occluders (preferably front to back, big objects first), objects ("occludees") can be tested for visibility. Graphics hardware allows to test multiple objects efficiently but there is a big problem. Because all rendering is buffered heavily, the information if some object is visible will be delayed for a long time (up to multiple frames). This is unacceptable as this either means very bad stalls (frame-rate hitches), bad frame-rate in general or objects being invisible for a while where they shouldn't be.

On some hardware/drivers this latency problem is less severe than on others but a delay of about one frame is about the best as it gets. That also means we cannot do efficient hierarchical tests efficiently, for example if some enclosing box is visible and then doing more fine grained tests with sub-parts. The occlusion test functionality is implemented in the engine and currently used for ocean rendering. We even use the amount of visible pixels to scale the reflection update frequency. Unfortunately, we also can have the situation where the ocean is not visible for one or two frames because of rapid view position changes. This happened for example in Crys...