



*n*VIDIA®

# Occlusion (HP and NV Extensions)

Ashu Rege

# Overview

- **What are the HP and NV occlusion tests**
- **Why should you care**
- **How do you use them**
- **Example – Incremental Object-level Culling**
- **Example – Order Independent Transparency**
- **Example – Lens Flares, Halos, etc.**
- **Example – Stencil Shadow Volumes**

# HP Occlusion Test (1)

- Extension name: HP\_occlusion\_test
- Provides a mechanism for determining “visibility” of a set of geometry
- After rendering geometry, query if any of the geometry **could** have or **did** modify the depth buffer.
- If occlusion test returns **false**, geometry could not have affected depth buffer
- If it returns **true**, it could have or did modify depth buffer

# HP Occlusion Test (2)

- All this verbiage basically means that your object is **not visible** if the test **fails** (returns false)
- It is **visible** if the test **passes** (returns true) in “usual” circumstances
- That is, e.g., you haven’t turned off the depth mask or color mask
- Typical usage: Render bounding box for target geometry. If test fails, you can skip the geometry altogether.

# HP Occlusion Test – How to Use

- **(Optional) Disable updates to color/depth buffers**

`glDepthMask(GL_FALSE)`

`glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)`

- **Enable occlusion test**

`glEnable(GL_OCCLUSION_TEST_HP)`

- **Render (bounding) geometry**

- **Disable occlusion test**

`glDisable(GL_OCCLUSION_TEST_HP)`

- **Read occlusion test result**

`glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP, &result)`



# HP Occlusion Test - Limitations

- Returns a simple TRUE or FALSE
- Often useful to know **how many** pixels were rendered
- Uses a “stop-and-wait” model for multiple tests
- Driver has to stop and wait for result of previous test before beginning next test
- Mediocre performance for multiple tests
- Eliminates parallelism between CPU and GPU

# NV Occlusion Query (1)

- Extension name: NV\_occlusion\_query
- Solves problems in HP\_occlusion\_test
- Returns **pixel count** – the no. of pixels that pass
- Provides an interface to issue **multiple queries at once** before asking for the result of any one
- Applications can now overlap the time it takes for the queries to return with other work increasing the parallelism between CPU and GPU



# NV Occlusion Query – How to Use (1)

- (Optional) Disable Depth/Color Buffers
- (Optional) Disable any other irrelevant non-geometric state
- Generate occlusion queries
- Begin  $i^{\text{th}}$  occlusion query
- Render  $i^{\text{th}}$  (bounding) geometry
- End occlusion query
- Do other CPU computation while queries are being made
- (Optional) Enable Depth/Color Buffers
- (Optional) Re-enable other state
- Get pixel count of  $i^{\text{th}}$  query
- If (count > MAX\_COUNT) render  $i^{\text{th}}$  geometry



# NV Occlusion Query – How to Use (2)

- **Generate occlusion queries**

```
GLuint queries[N];  
GLuint pixelCount;  
glGenOcclusionQueriesNV(N, queries);
```

- **Loop over queries**

```
for (i = 0; i < N; i++) {  
    glBeginOcclusionQueryNV(queries[i]);  
    // render bounding box for ith geometry  
    glEndOcclusionQueryNV();  
}
```

- **Get pixel counts**

```
for (i = 0; i < N; i++) {  
    glGetOcclusionQueryuivNV(queries[i], GL_PIXEL_COUNT_NV,  
                            &pixelCount);  
    if (pixelCount > MAX_COUNT)  
        // render ith geometry
```

```
}
```



# Example – Incremental Object-Level Culling

- Previous example was rather naïve
- Rendering a bounding box will cost you fill
- Need to be more intelligent in how you issue queries!
- Big win if you use query for object that you were going to render in any case
- If you have too many objects and want to avoid generating too many queries, you can skip the occlusion queries for **visible** objects for the next few frames. (For occluded objects you still have to test in case it has become visible.)

## Incremental Object-Level Culling (2)

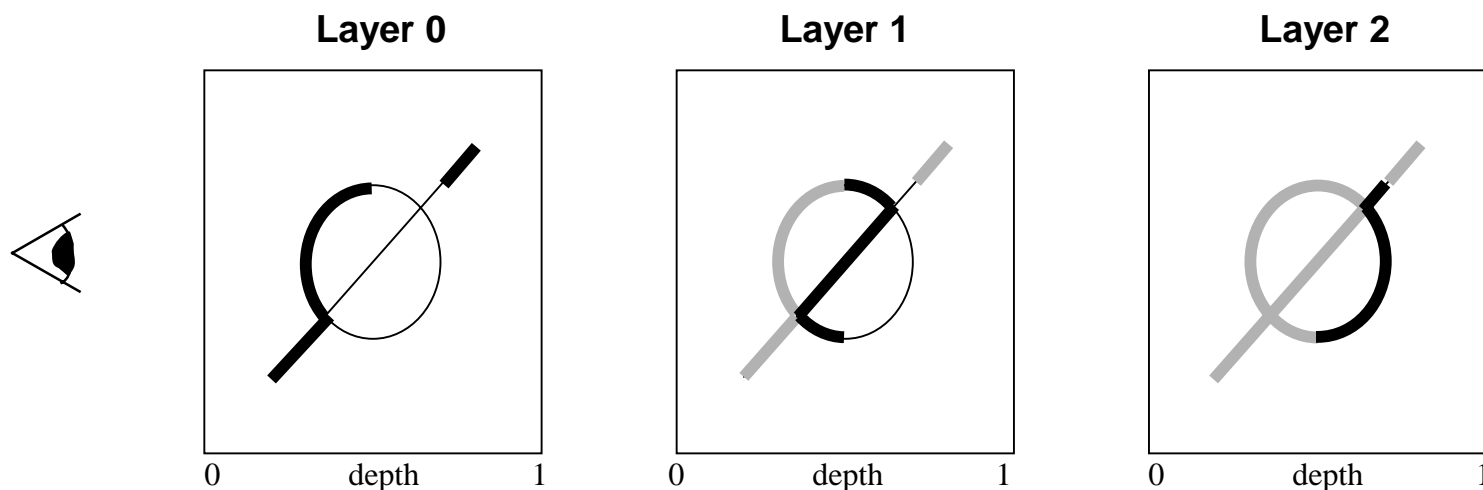
- Better approach for multi-pass algorithms:
- Render scene (roughly) from front to back
- Make sure your big occluders like walls etc. are drawn first
- Issue queries for other objects in the scene
- If query returns 0 in first pass, you can skip the object in subsequent passes
- If object is occluded it will get eliminated at worst in the second pass. (Might pass test in first pass because of sorting)
- Works great if your first pass is used to set up the depth buffer only. (Turn off everything but...) This will get even faster on future hardware.



## Example – Order Independent Transparency (OIT)

- **Cool approach to transparency!**
- **See whitepaper by Cass Everitt on developer site for details**
- **Hardware does the sorting for you**
- **Can be ‘bolted on’ to applications very easily**
- **Uses ‘depth peeling’ to sort the layers in the scene**

# OIT – Depth Peeling



- Strip away depth layers with each pass
- Capture RGBA after each layer is peeled
- Blend from back to front
- *How many layers to use?* Use occlusion query with some pixel count (e.g. as % of screen size) to determine when to stop!

# Example – Lens Flares, Halos,... (1)

- In typical algorithm lens flare is applied last
- A 2D texture map is rendered as a billboard
- Occlusion is dealt with by first rendering the objects in the scene including the light source
- Determining whether the light source is visible or occluded is typically done by using `glReadPixels` of the depth buffer
- To determine *how much* of the light source is visible you would grab the region of the screen that corresponds to the light source and get the ratio of the visible to total pixels



## Example – Lens Flares, Halos,... (2)

- Problems with this approach:
- To read back contents of the scene, application must wait for rendering to complete, i.e., the pipeline must be flushed – requires CPU wait
- Reading **back** data is slow and done at PCI speeds
- Solution: Use NV\_occlusion\_query of course!
- Pixel count will tell you what % of the light source is visible to get an approximation on intensity
- The query is **asynchronous** so CPU can proceed with other computation during the query computation

# Example – Stencil Shadow Volumes

- Typical algorithm: (for a given light source)
- Generate b-rep's for shadow volumes of occluders
- Determine whether pixel is shadowed or not using depth and stencil buffers
- Use `NV_occlusion_query` to eliminate occluders:
- If the bounding box of an occluder returns a pixel count of 0, it cannot produce visible shadows so you can eliminate it from consideration
- One issue to bear in mind: The **fill** for a bbox might make eliminating occluders more expensive. One solution might be to use lower resolution depth buffer



# Other Examples

- **For determining whether a light with falloff that is outside the view should be considered in shadow computations.**
- **Associate geometry with lit region. Determine if geometry is occluded or not to tell you if light is relevant**
- **Use in conjunction with L.O.D. rendering**

# Conclusion

- Simple to use in applications yet very powerful
- Provides **pixel count** – a very useful quantity
- Can be used asynchronously, more parallelism for you and me
- Versatile extension that can be used in a wide variety of algorithms/applications

# Questions, comments, feedback?

---

- **Ashu Rege, arege@nvidia.com**
- **[www.nvidia.com/developer](http://www.nvidia.com/developer)**