Additional
Rendering Considerations

Topics

- Forward rendering
- Object culling
- Light culling
- Dynamic light loops

Forward Rendering

Forward rendering

- The rendering technique we've been using
- Calculating radiance* for a pixel as geometry is being drawn
  - *Radiance: emitted or reflected light
- Required information:
  - Geometry: position, normal, etc.
  - Material: shininess, color, etc.
  - Light: color, direction, etc.

Forward rendering: Pros

- Easy to implement

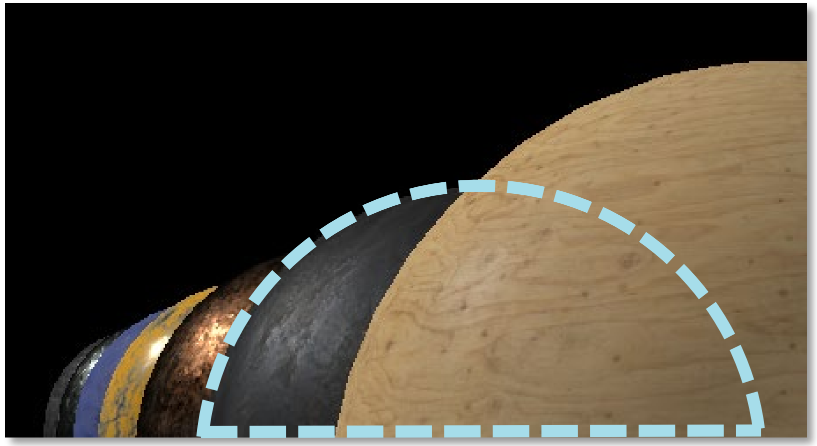
- Makes a lot of sense
  - Gotta run a pixel shader – might as well do lighting there!

- Works with opaque & transparent objects
  - As long as transparent objects are rendered after all opaque objects
  - And transparent objects are sorted back to front

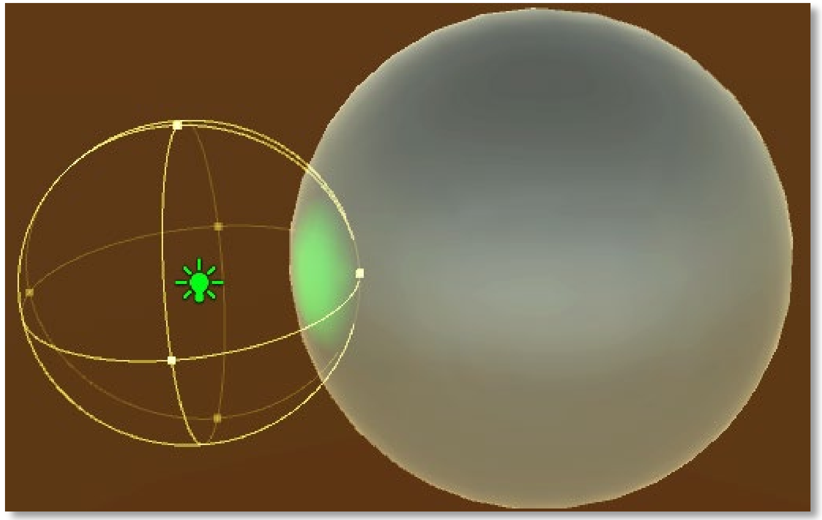
- Works with hardware anti-aliasing (MSAA)

Forward rendering: Cons

- Overdraw* is expensive
  - *Drawing a pixel, then covering it
  - Later within the same frame



- Lighting calculated for entire objects
  - Even when lights have very small ranges



- Lighting complexity is
  - $O(number\ of\ objects\ * number\ of\ lights)$

Alternatives to forward rendering

- Deferred rendering
  - Defers lighting until after all geometry is rendered
  - Requires multiple render targets
- Forward+ rendering
  - Divides screen into very small tiles (8x8 pixels)
  - Process lights whose bounds overlap a tile
  - Usually involves compute shaders
- Advanced topics requiring different C++ render architecture

Improvements to forward rendering

- $O(objects * lights)$ → Reduce objects or reduce lights!

- Cull objects
  - Check camera bounds (frustum) in C++
  - Only draw if object is visible
  - A bounding volume hierarchy (octree or similar) is useful here

- Cull lights
  - Determine which lights might actually affect an object
  - Only pass those lights to the shader

Culling Objects

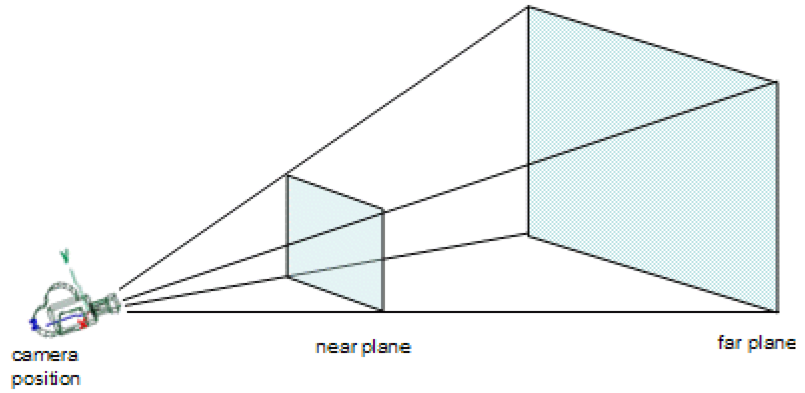
Culling objects

- Skipping objects we can’t see – *before* we render them!

- If it won’t be seen this frame…
  - Don’t even attempt to render it
  - Saves us from even having to transform the vertices

- Multiple ways of handling this
  - Per-object checking
  - Grid-based checking
  - Bounding volume hierarchy (BVH)

Culling & the camera



- The camera frustum
  - The bounds of the camera
  - Defined by the projection, specifically

- Anything inside the frustum can be seen
- Anything outside the frustum cannot

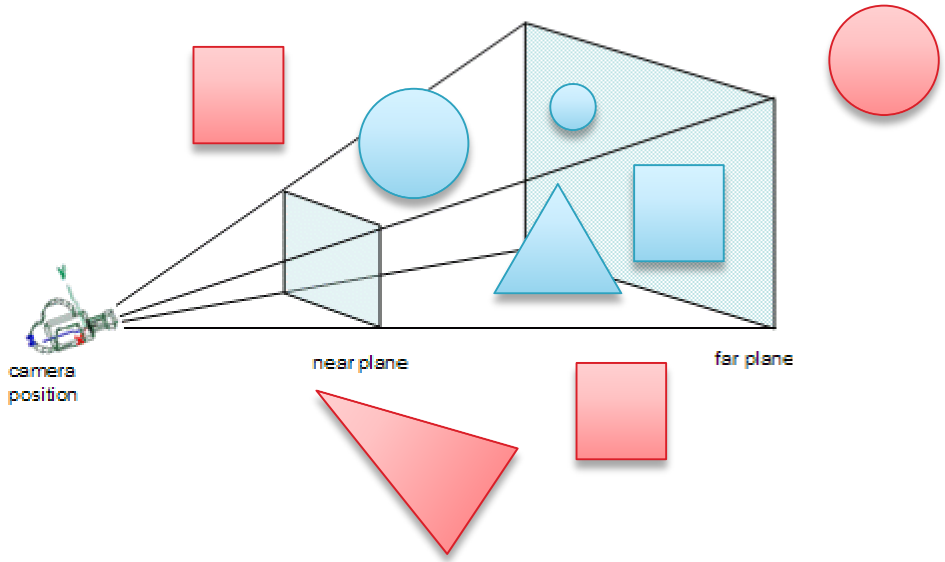
- If we can detect objects inside/outside in C++
  - We can decide if we even need to render

Culling requires entity bounds

- Need to know an entity's world-space bounds
- Mesh could store simple AABB bounding box
  - But we need more than this
  - This only tells us the size of the mesh before transformations
- Needs to take into account entity's transform!
  - Transform bounds by world matrix → Oriented Bounding Box (OBB)
  - Or build new AABB from transformed AABB corners

Culling: Per-object

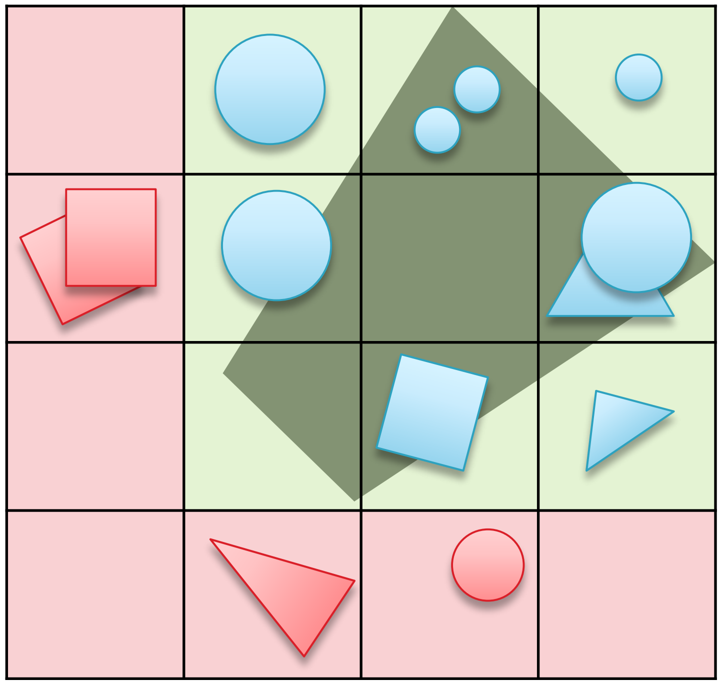
- Compare each object's bounds to the frustum
  - Box/Frustum intersection test



- Easy but somewhat slow: O(n)

Culling: Grid-based

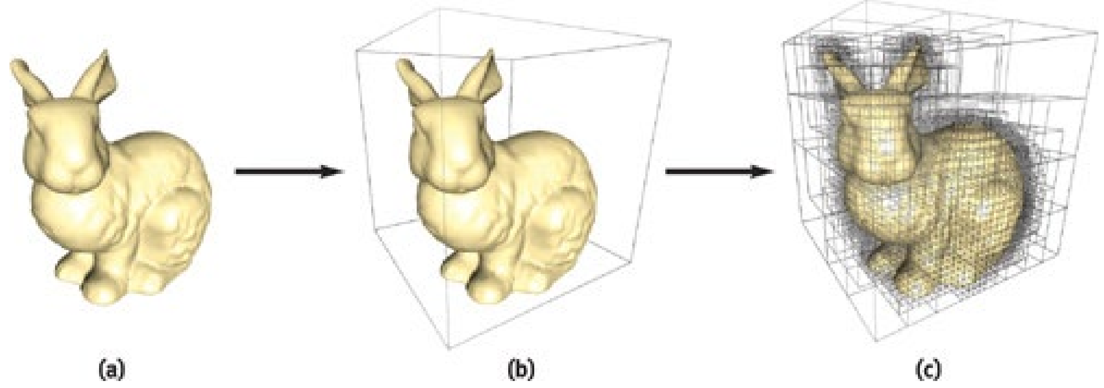
- Divide scene into grid (3D array)
  - Grid holds all objects
  - Test grid cell against frustum
  - Draw any objects in a “valid” cell



- Usually faster if cells hold >1 object
  - Can result in false positives!

Culling: Bounding Volume Hierarchy (BVH)

- A hierarchy (tree) of bounding volumes (chunks of space)
  - Quadtree (2D)
  - Octree (3D)
  - Binary space partitioning tree (3D)
- Similar to grid-based, but with a tree
- Generally fastest: O(log n)
  - Also trickiest to implement



When do we cull?

- Beginning of Draw()
  - Camera & all entities need to update first
  - Hopefully all bounding boxes are updated at this point

- Render loop can perform intersection tests
  - Only render entities that “pass the test”

- Alternatively, loop through everything first
  - Collect list of objects that need to be rendered
  - Can perform other useful operations (sorting?) on this list

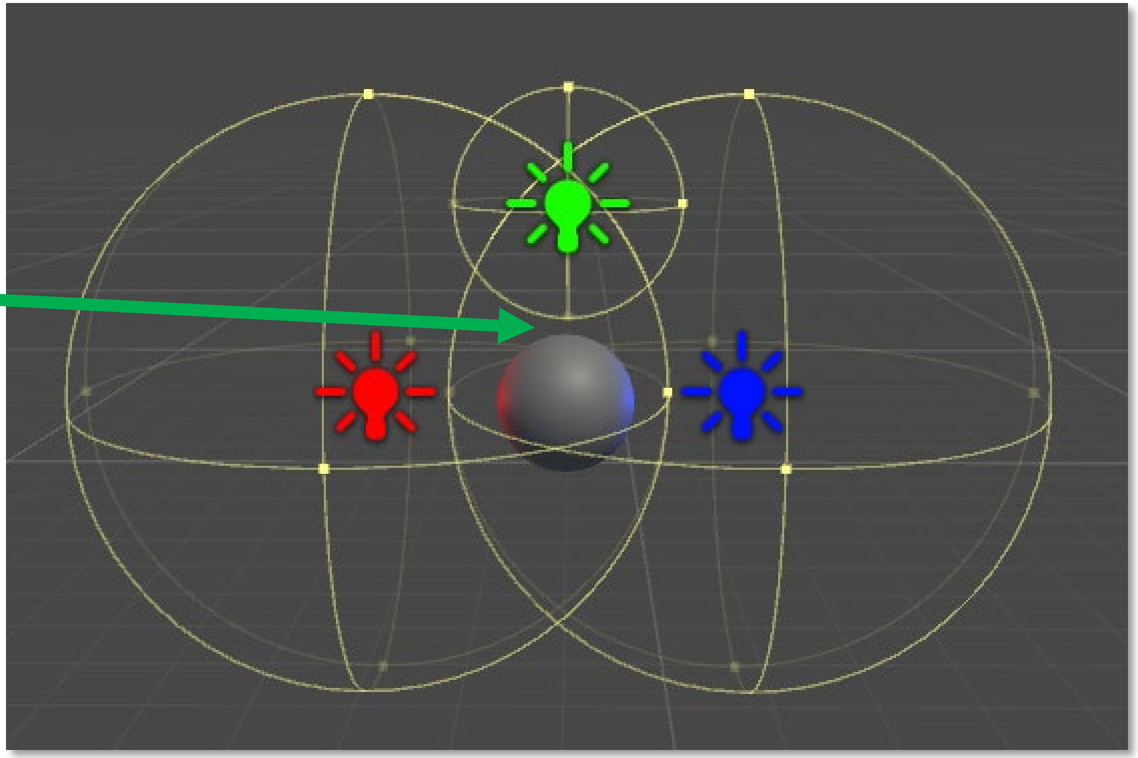
Culling Lights

Light culling

- Don’t process lights that won’t affect objects
  - No sense in doing unnecessary work

Green light doesn’t intersect sphere.

No need to even send to the shader.



Light culling

- Culling occurs in C++
  - Requires range-based point/spot attenuation

- Determine which lights affect objects:
  - **Point**: Based on light's range
  - **Spot**: Based on range, cone size and direction
  - **Directional**: These hit *everything*

- Requires each entity to know its bounds
  - Mesh could store simple AABB bounding box
  - Needs to take into account entity's transform!

Light culling – What to send to GPU?

- Before rendering each object:
  - Loop through lights
  - Check each point & spot light against object bounds
  - Add overlapping lights to *intersecting lights* list
- Only send *intersecting lights* to shader
  - **Light array** only holds lights that might actually hit object
  - **Light count** is number of lights in *intersecting lights* list
- Requires a light loop in HLSL that iterates based on CPU data

Is light culling worth it?

- Is this faster than just letting the GPU chug through lights?
  - Maybe? Maybe not? Depends on your implementation!
- What is the cost of each light in HLSL?
  - More expensive BRDFs take more time
- What is the cost of light culling?
  - Brute force is slower than using a good bounding volume hierarchy
- Test and find out for your specific application

Dynamic Light Loop

In a shader

Dynamic light loop

- Process a dynamic # of lights in a shader
  - Loop through an array of lights
  - Add the results of each light calculation

- This is similar to what we've discussed already

- Except with a *dynamic* number of lights
  - Don't hardcode light1, light2, etc.
  - Don't assume there are always *N* lights

Dynamic light count – Light array

```
// Define the max number of lights
#define MAX_LIGHTS 128

cbuffer externalData : register(b1)
{
    // An array of lights (const size)
    LightStruct lights[MAX_LIGHTS];

    // Amount of lights to process THIS FRAME
    int lightCount;
}
```

Example light struct setup

```
#define LIGHT_TYPE_DIR     0
#define LIGHT_TYPE_POINT   1
#define LIGHT_TYPE_SPOT    2

// Make a matching struct on CPU (C++) side
struct LightStruct
{
      int    Type;
      float3 Direction;    // 16 bytes
      float  Range;
      float3 Position;     // 32 bytes
      float  Intensity;
      float3 Color;        // 48 bytes
      float  SpotFalloff;
      float3 Padding;      // 64 bytes
};
```

Light loop pseudo-code

```
float3 c = float3(0,0,0); // Total color

// Loop through all lights THIS FRAME
// (Assumes lightCount < MAX_LIGHTS)
for (int i = 0; i < lightCount; i++)
{
   switch (lights[i].Type)
   {
     case LIGHT_TYPE_DIR:   c += DirLight(…);   break;
     case LIGHT_TYPE_POINT: c += PointLight(…); break;
     case LIGHT_TYPE_SPOT:  c += SpotLight(…);  break;
   }
}
```

Dynamic light loop: Pros & cons

- This is the method I usually use

- Pros:
  - Relatively simple to implement
  - Flexible amount of lights (up to the max)

- Cons:
  - Branches (luckily each pixel takes the same branch – best case)
  - One fairly expensive shader – overdraw is *rough*
  - Lighting complexity is still $O(objects * lights)$