Robust Stencil Shadow Volumes

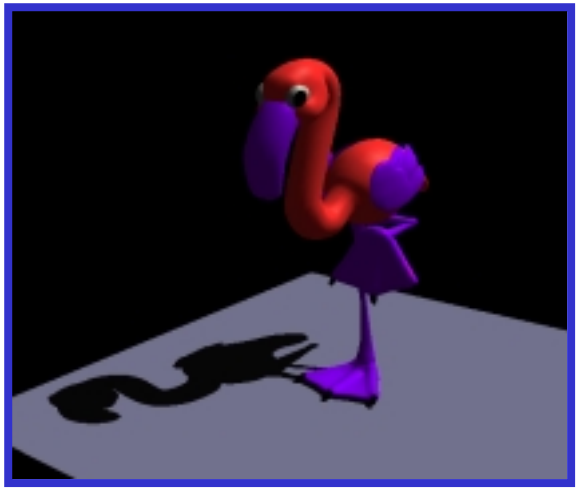
CEDEC 2001
Tokyo, Japan

Mark J. Kilgard
Graphics Software Engineer
NVIDIA Corporation

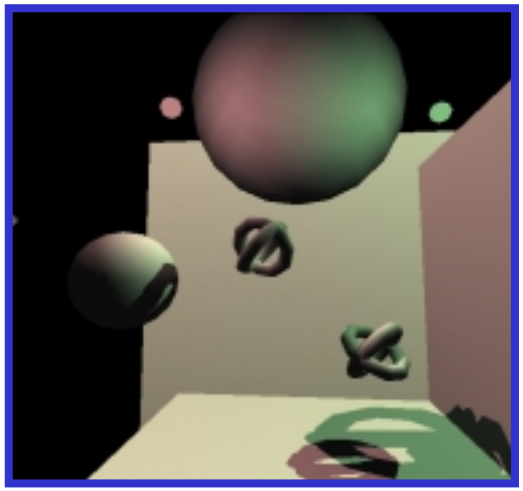
Games Begin to Embrace Robust Shadows

- John Carmack's new Doom engine leads the way
  - First-class lights
  - Vital to the look of the engine
- Cinematic shadowing effects
  - Scary scenes possible

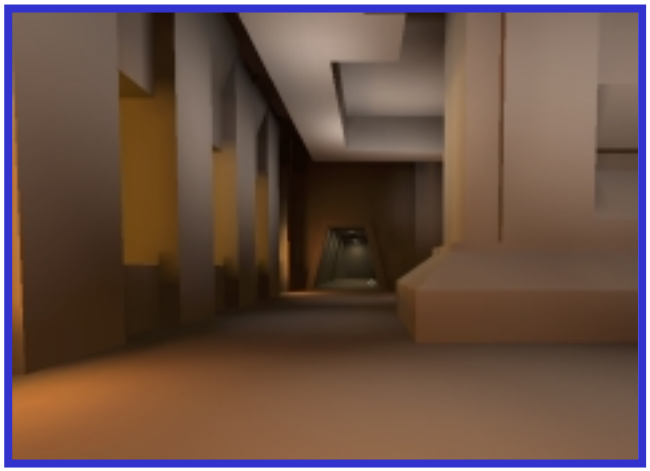
Variety of Shadow Techniques



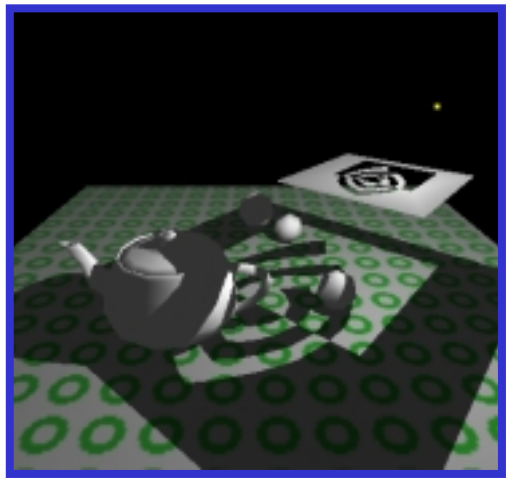
Projected planar shadows



Shadow volumes



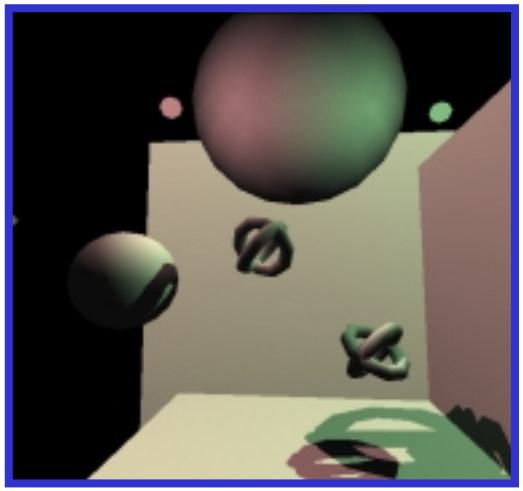
Light maps



Hybrid approaches

Stenciled Shadow Volumes

Technique that Carmack is using for new Doom engine



Shadow volumes

Advantages

- Support omnidirectional lights
- Exact shadow boundaries

Disadvantages

- Fill-rate intensive
- Expensive to compute shadow volumes
- Hard shadow boundaries, not soft shadows
- Difficult to implement robustly

NVIDIA

Let Us Consider Omni-directional Shadowing

- **Situation: a light source centered in a room**
  - **Dynamic characters in the room**
  - **Everything should shadow everything**
- **This is a situation for stenciled shadow volumes**

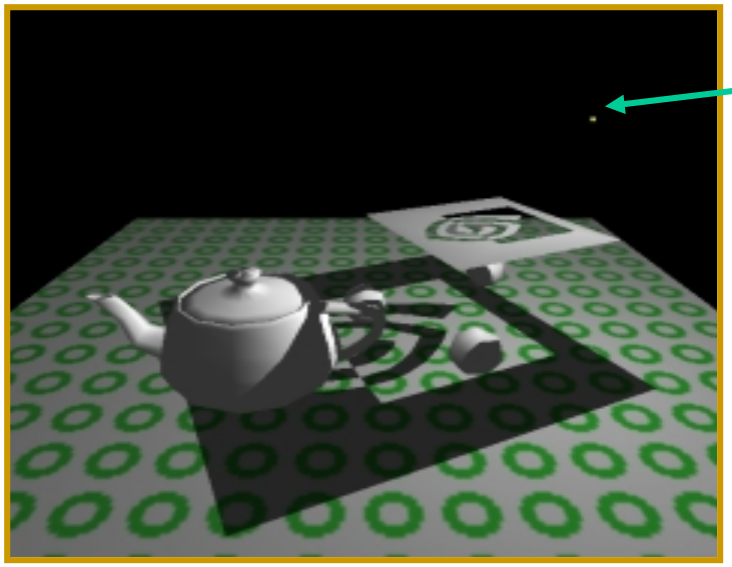
Review: Stenciled Shadow Volumes

- A single point light source splits the world in two
  - Shadowed regions
  - Unshadowed regions
  - Volumetric shadow technique
- A shadow volume is the boundary between these shadowed and unshadowed regions
  - Determine if an object is inside the boundary of the shadowed region and know the object is shadowed
- First described by [Crow 77]

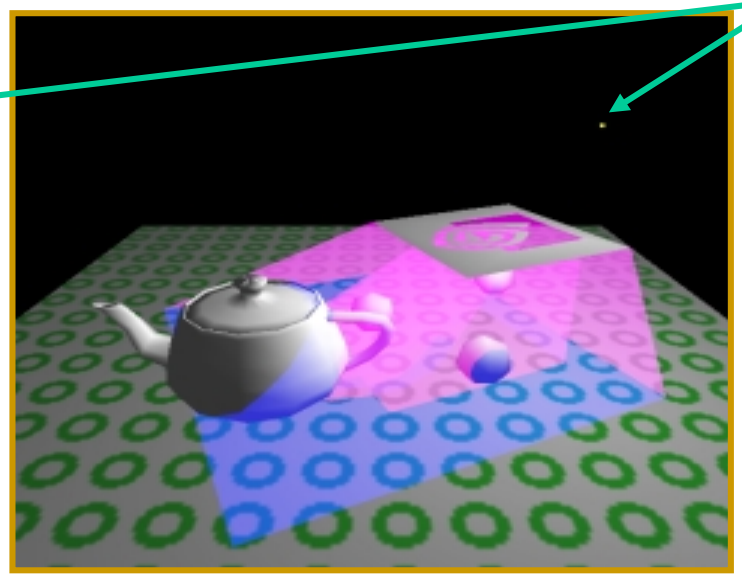
By the way, Frank Crow works at NVIDIA now

Visualizing Shadow Volumes

- Occluders and light source cast out a shadow volume
  - Objects within the volume should be shadowed



Scene with shadows from an NVIDIA logo casting a shadow volume



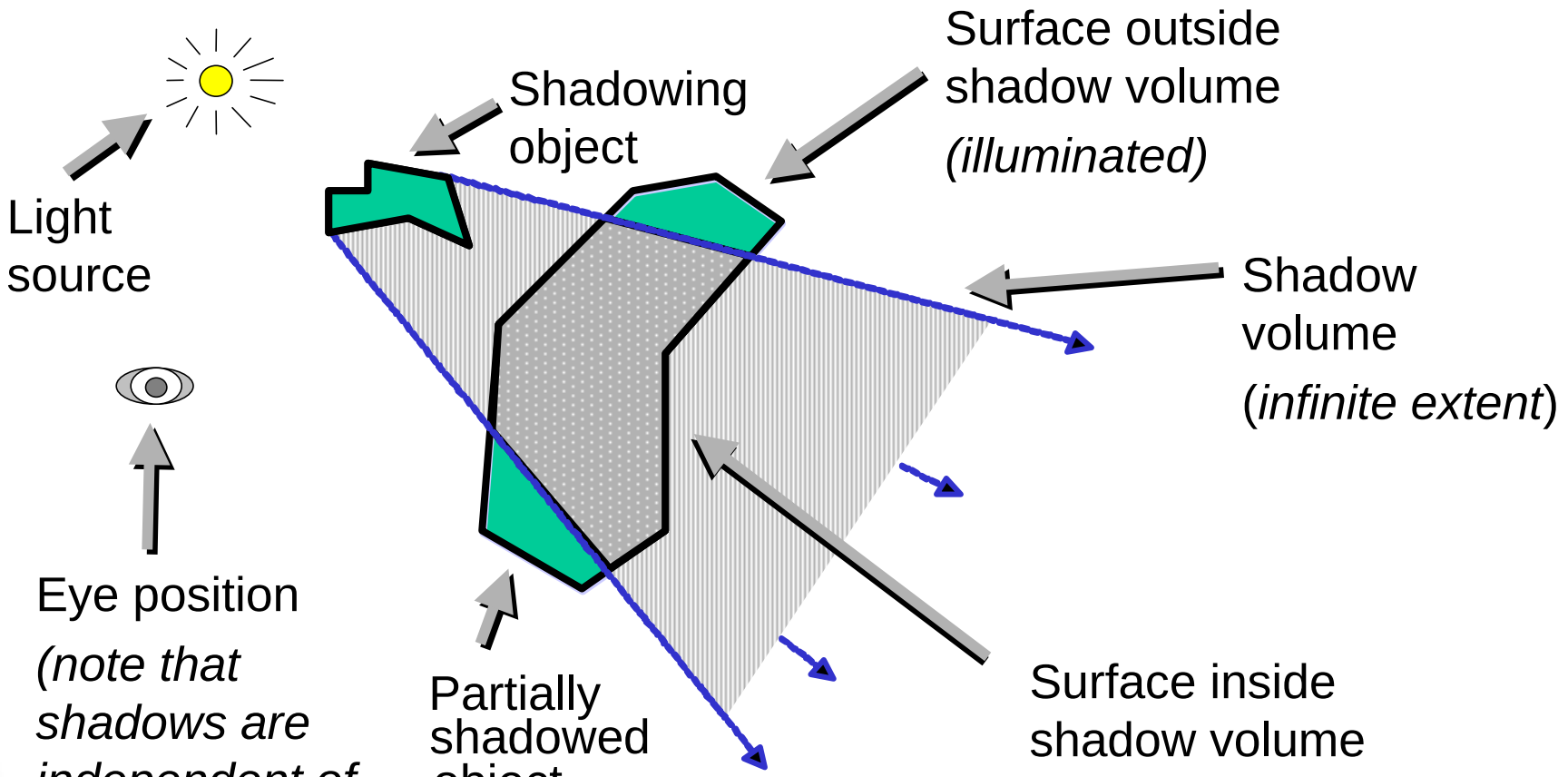
Visualization of the shadow volume

NVIDIA

Shadow Volume Algorithm

- **High-level view of the algorithm**
  - **Given the scene and a light source position, determine the shadow volume (harder than it sounds)**
  - **Render the scene in two passes**
    - **Draw scene with the light *enabled*, updating *only* fragments in *unshadowed* region**
    - **Draw scene with the light *disabled*, updated *only* fragments in *shadowed* region**
  - **But how to control update of regions?**

2D Cutaway of a Shadow Volume



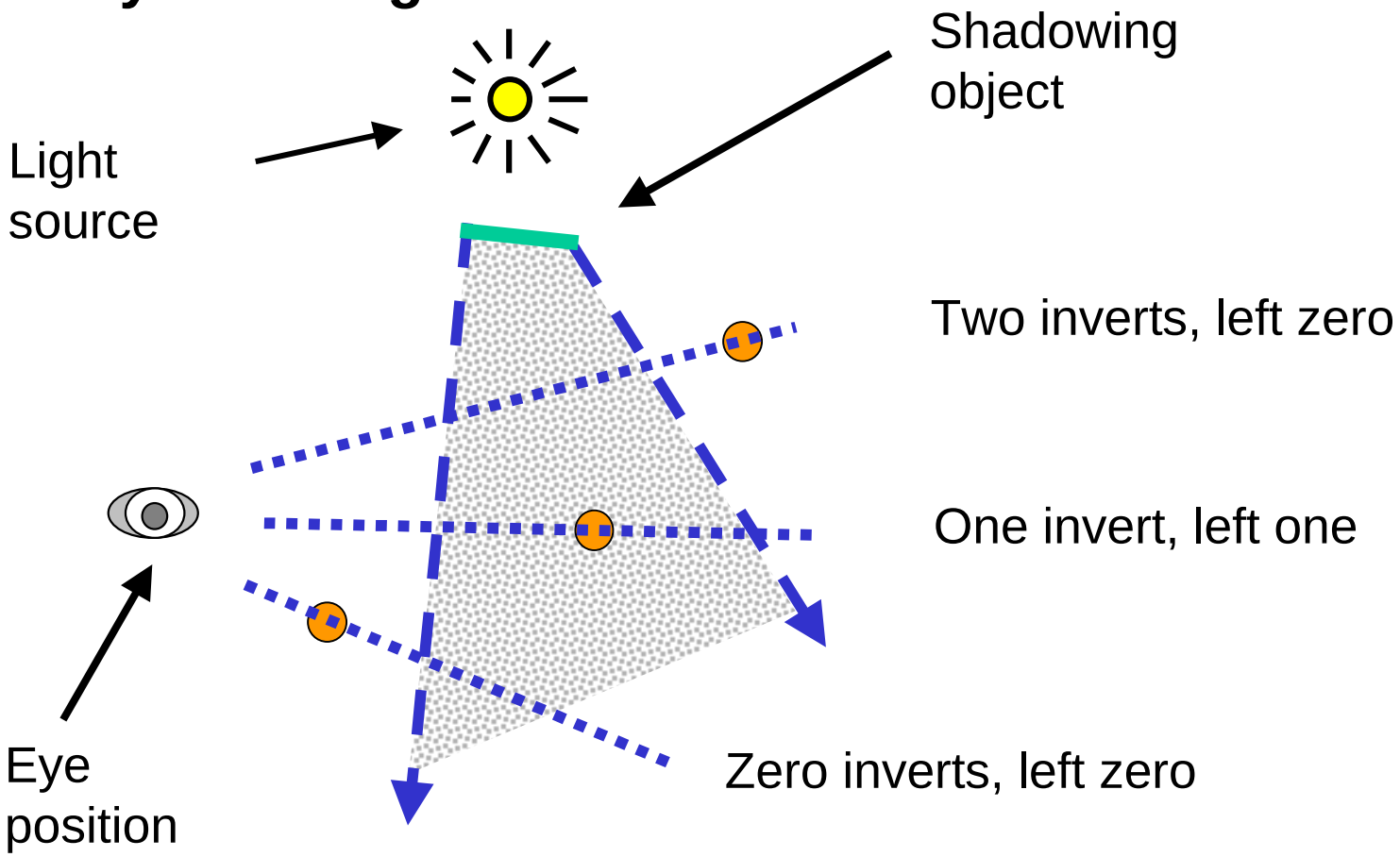
NVIDIA

Tagging Pixels as Shadowed or Unshadowed

- **High-level algorithm does not say how to update only either pixels in or out of the shadow volume!**
- **The stenciling approach**
  - **Clear stencil buffer to zero and depth buffer to 1.0**
  - **Render scene to leave depth buffer with closest Zs**
  - **Render shadow volume into frame buffer with depth testing but *without* updating color and depth, but *inverting* a stencil bit**
  - **This leaves stencil bit set within shadow!**

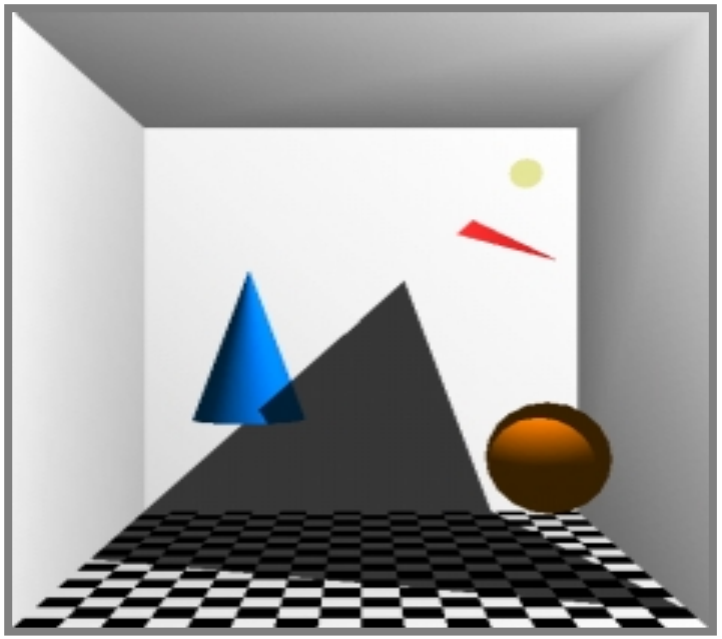
Stencil Inverting of Shadow Volume

- **Why inverting stencil works**

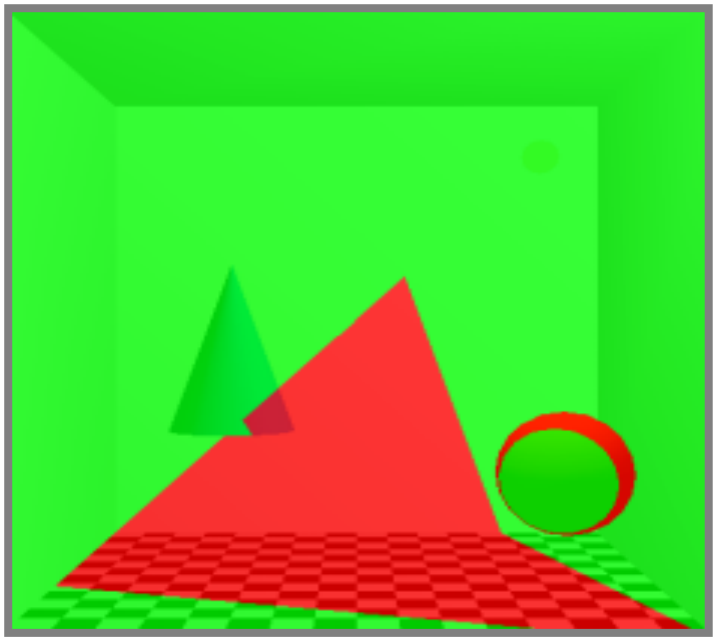


Visualizing Stenciled Shadow Volume Tagging

Shadowed scene



Stencil buffer contents



red = stencil value of 1
green = stencil value of 0

GLUT *shadowvol* example credit: Tom McReynolds

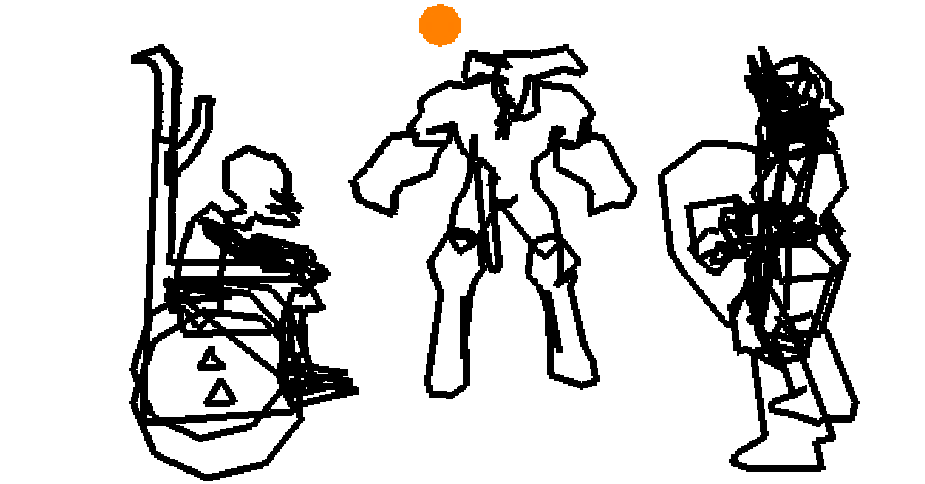
Computing Shadow Volumes

- Harder than you might think
  - Easy for a single triangle, just project out three infinite polygons from the triangle, opposite the light position
  - But shadow volume polygons should not intersect each other for invert trick to work
    - This makes things hard
  - For complex objects, projecting object's 2D silhouette is a good approximation (flat objects are easy)
  - Static shadow volumes can be pre-compiled

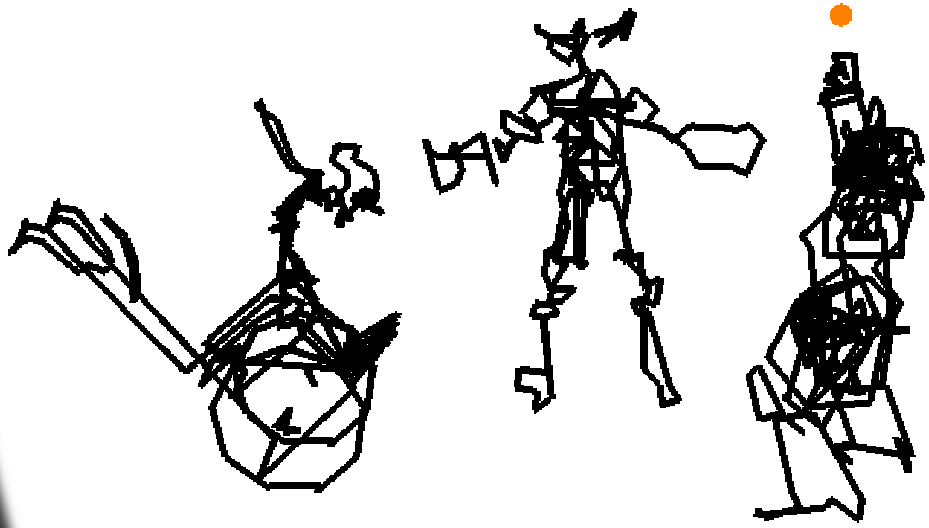
Computing Shadow Volumes For Polygonal Models

- **High-level: determine "possible silhouette" edges of the model**
  - **Transform light into object space**
  - **Compute the plane equation for every polygon in the model (can be pre-computed for static models)**
  - **For every polygon in the model, determine if the object-space light position is behind or in front of the polygon's plane**
    - **I.e., Is the planar distance from the polygon's plane to the light positive or negative?**
  - **Search for edges where polygons have opposite facingness toward the light**
    - **These edges are possible silhouette edges**

Examples of Possible Silhouette Edges for Models



An object viewed from the same basic direction that the light is shining on the object has an identifiable light-view silhouette



An object's light-view silhouette appears quite jumbled when viewed form a point-of-view that does not correspond well with the light's point-of-view

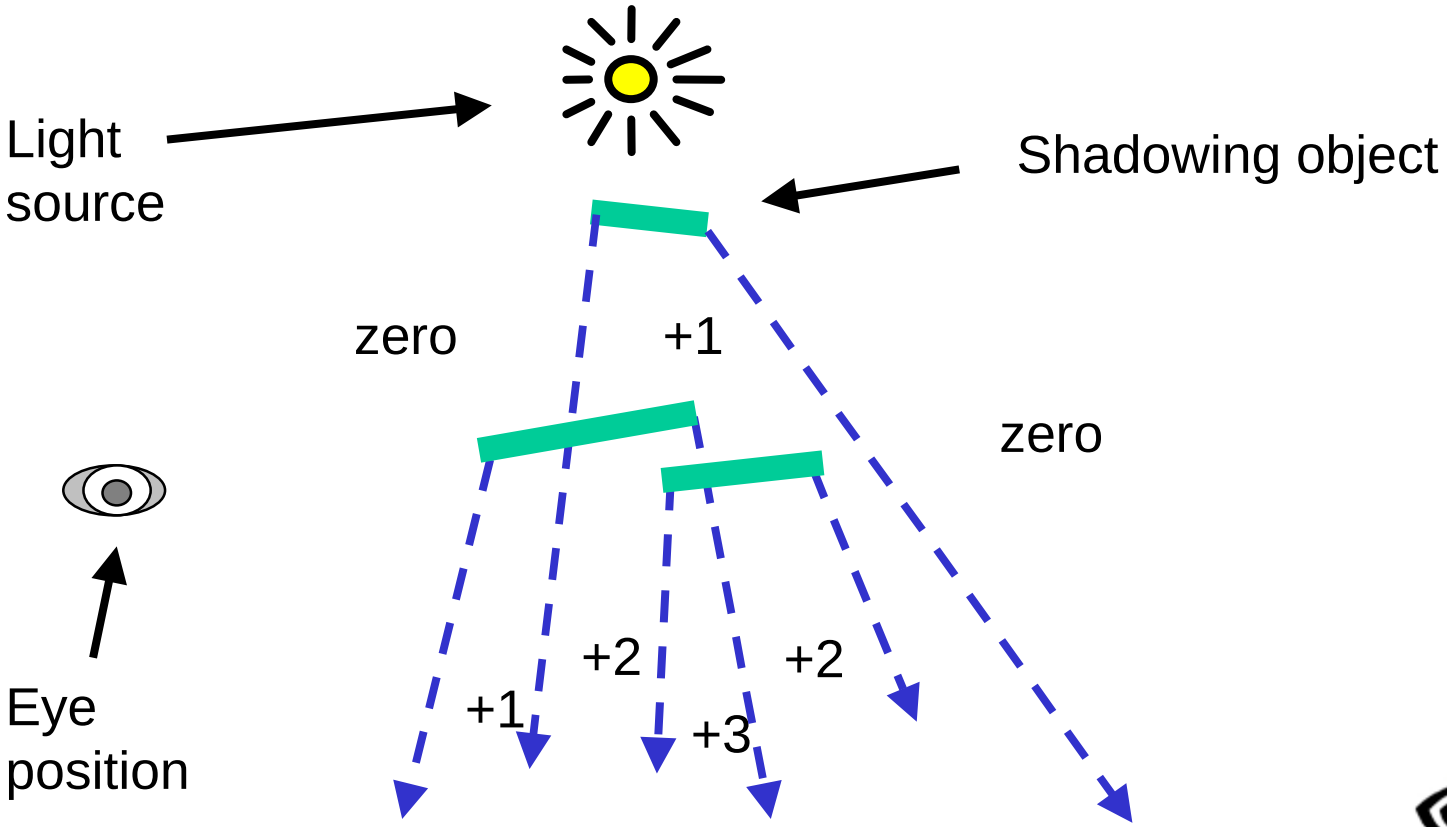
NVIDIA

For Shadow Volumes With Intersecting Polygons

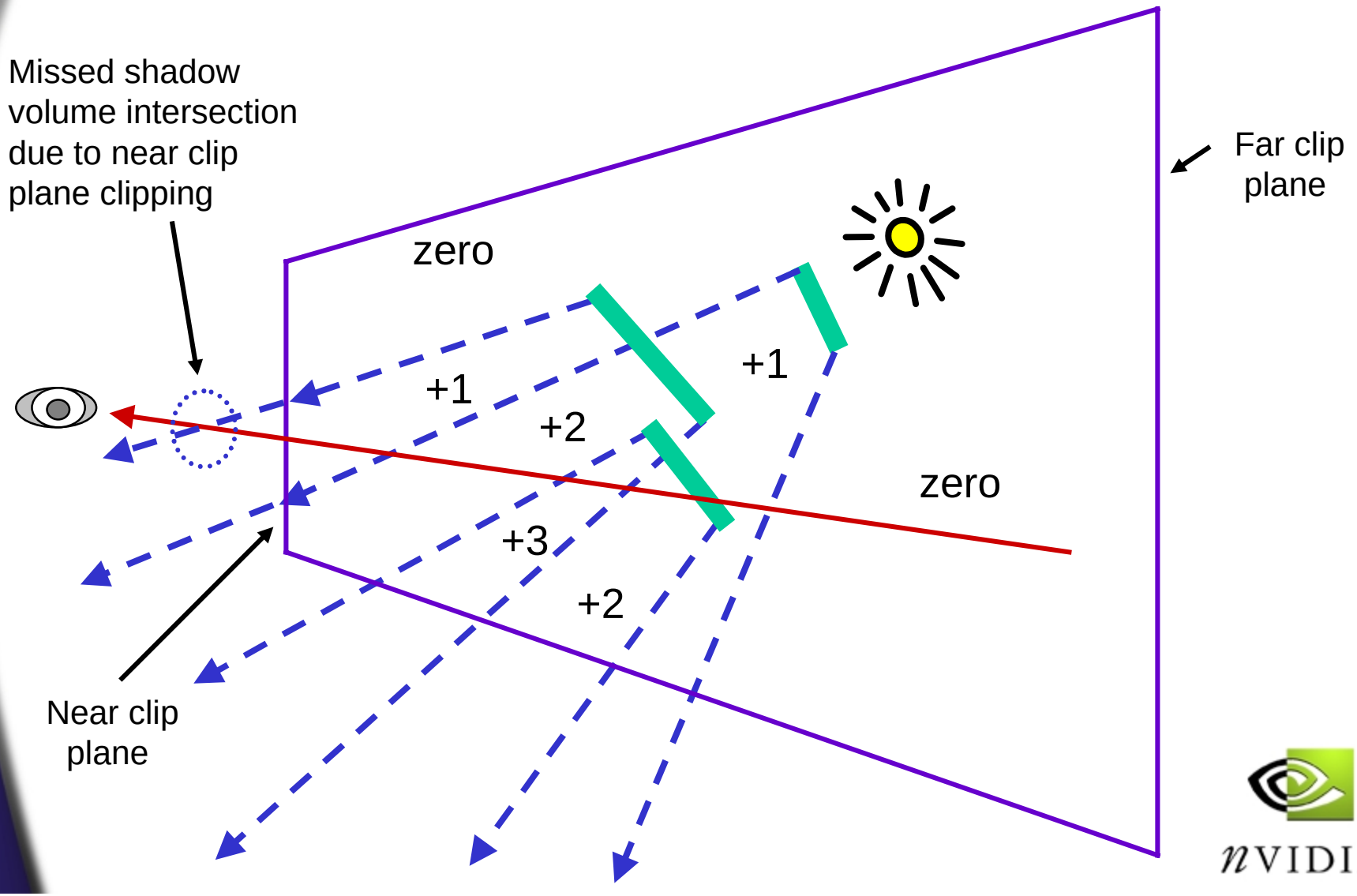
- Use a stencil enter/leave counting approach
  - Draw shadow volume twice using face culling
    - 1st pass: render ***front*** faces and ***increment*** when depth test passes
    - 2nd pass: render ***back*** faces and ***decrement*** when depth test passes
  - This two-pass way is more expensive than invert
  - And burns more fill rate drawing shadow volumes
  - Inverting is better if all shadow volumes have no polygon intersections (very rare)

Why Increment/Decrement Stencil Volumes Work

- **Example in 2D**

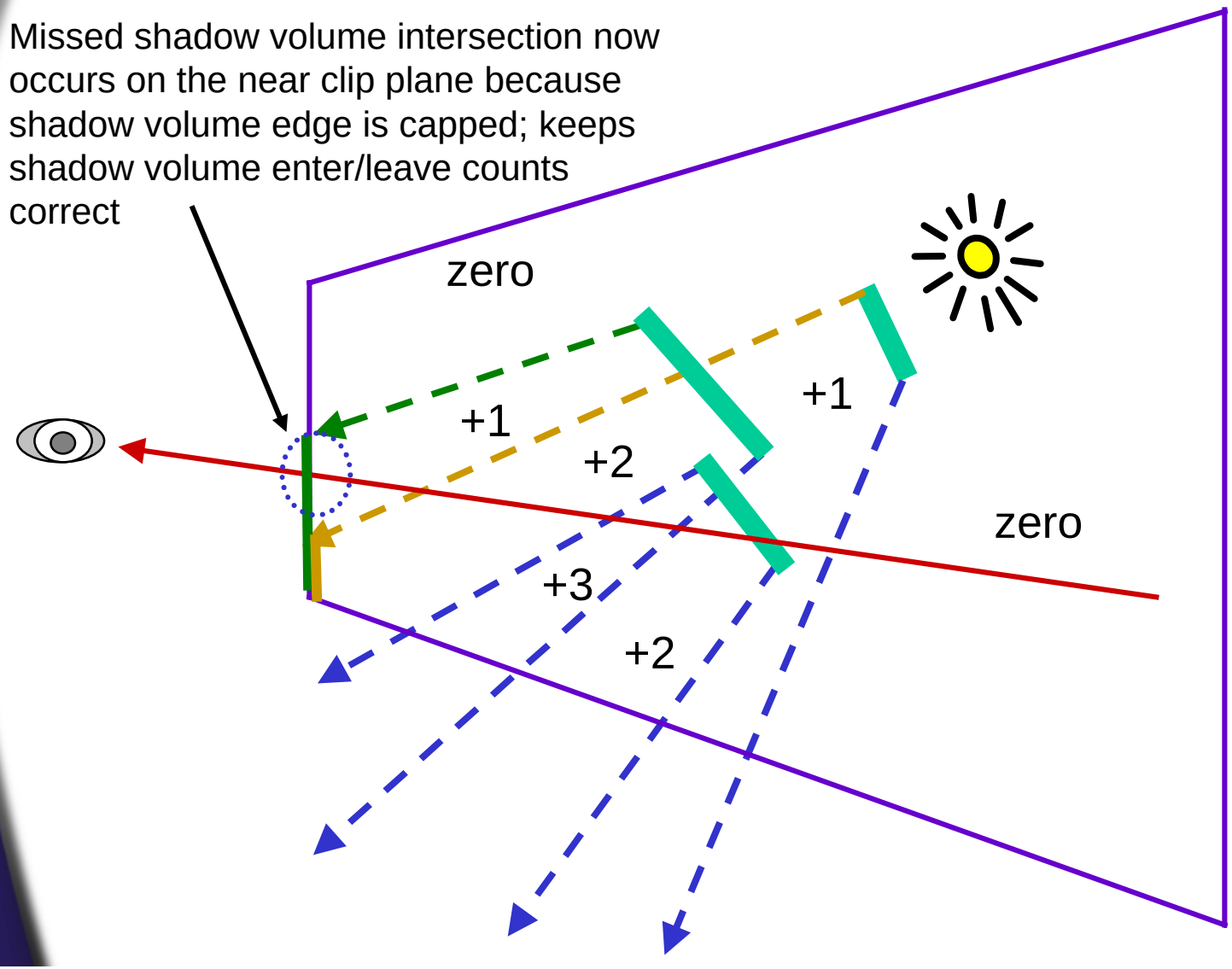


Problems with Near Plane Clipping



NVIDIA

Capping Shadow Volumes at The Near Clip Plane



3D version of the problem is harder than 2D case!

NVIDIA

Shadow Volume Near Plane Clipping Artifacts

- **Typically, shadow artifacts when light is behind objects**

Light source behind demon's head

Correct shadows:
Near plane capped properly

Incorrect shadows:
Near plane NOT capped

Incorrect shadows: Projection of silhouette edges to the near clip plane shows capping is responsible for the artifacts

Alternative to Near Plane Capping: Carmack's Reverse

- Conventional shadow volume stencil usage
  - Increment stencil when depth test passes for front facing shadow volume polygons
  - Decrement stencil when depth test passes for back facing shadow volume polygons
- John Carmack [1998?] reverses this usage
  - Increment stencil when depth test fails for back facing shadow volumes
  - Decrement stencil when depth test fails for front facing shadow volume polygons

Implications of Carmack's Reverse

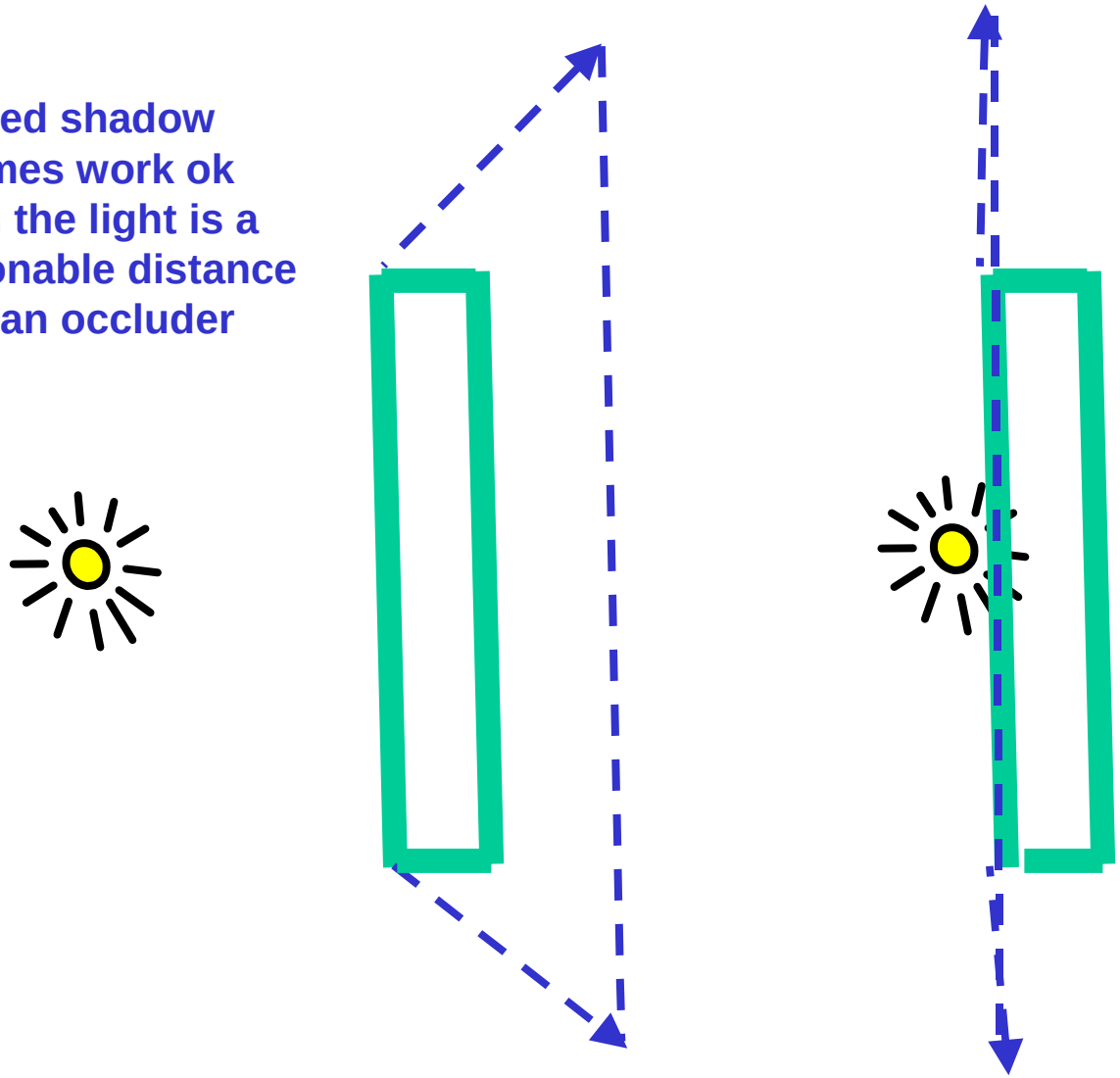
- No longer have to worry about near clip plane capping
- But the reverse shifts the problem to that the far clip plane clipping shadow volumes
  - Approach assumes that shadow volumes are truncated and capped after a certain finite distance that can never extend beyond the far clip plane
    - Typically ok for attenuated lights
  - Works ok because it is easier to move the far clip plane further out than move the near clip plane closer in

More Implications of Carmack's Reverse

- Requires always truncating the shadow volumes and capping them some distance out
- Setting the far clip plane far in the distance to avoid shadow volume intersections does compromise the available depth buffer precision
- A light very close to a shadowing polygon can cast a shadow so large that it fails to extend far enough

Polygons Very Close to the Light Source Cause Problems

Capped shadow volumes work ok when the light is a reasonable distance from an occluder



Capped shadow volumes eventually collapse into an increasingly marginal volume when the light gets extremely close

NVIDIA

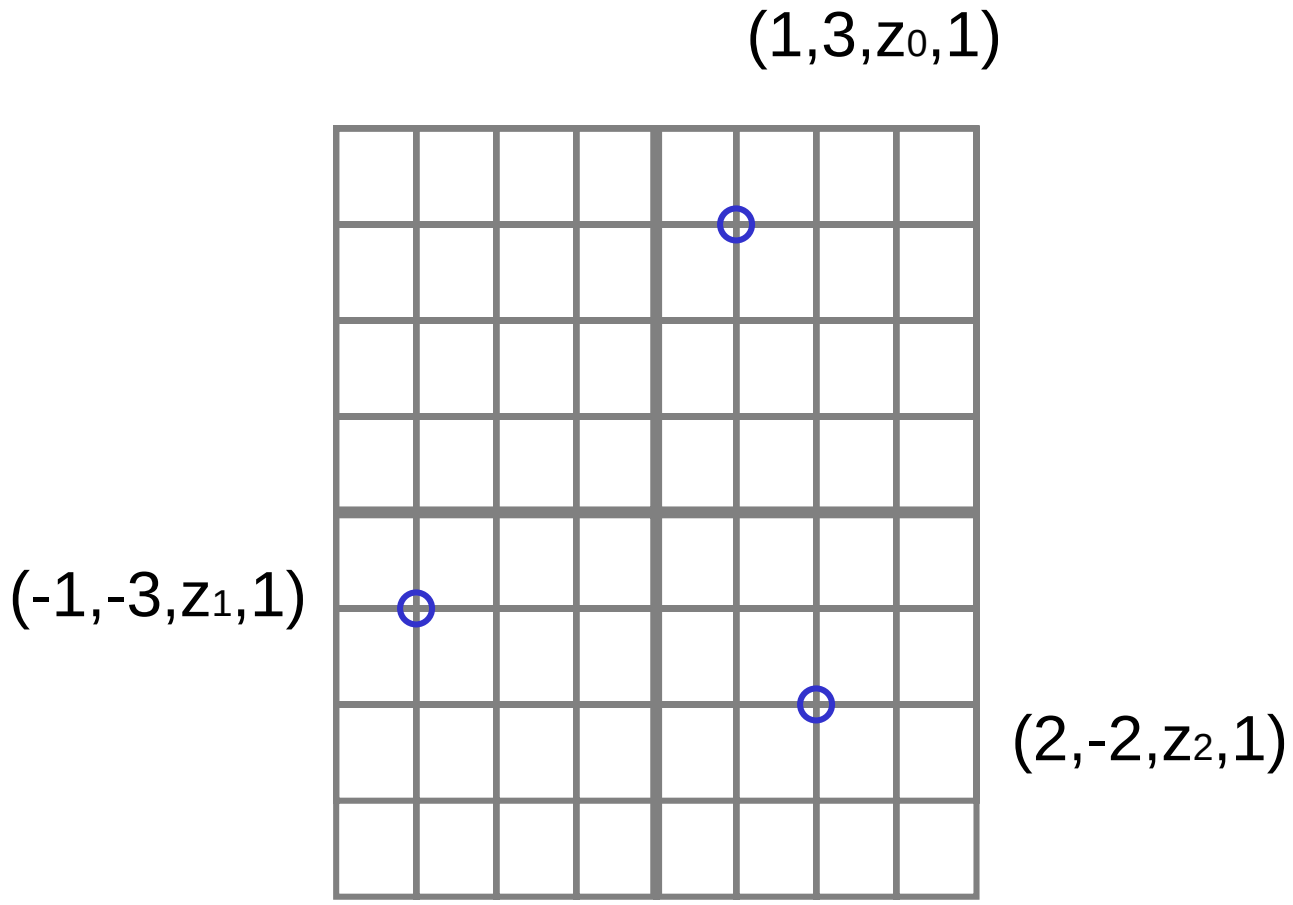
Carmack's Doom Solution

- Uses Carmack's reverse
- Support only attenuated lights
  - Attenuation means illumination can be bounded
  - Dynamically move far clip plane out to enclose light's maximum region of illumination
- Clip shadow volumes to eliminate "light too close to shadowing polygon" case
  - Extra clipping work
- Optimize spotlights
  - Exploits light's frustum

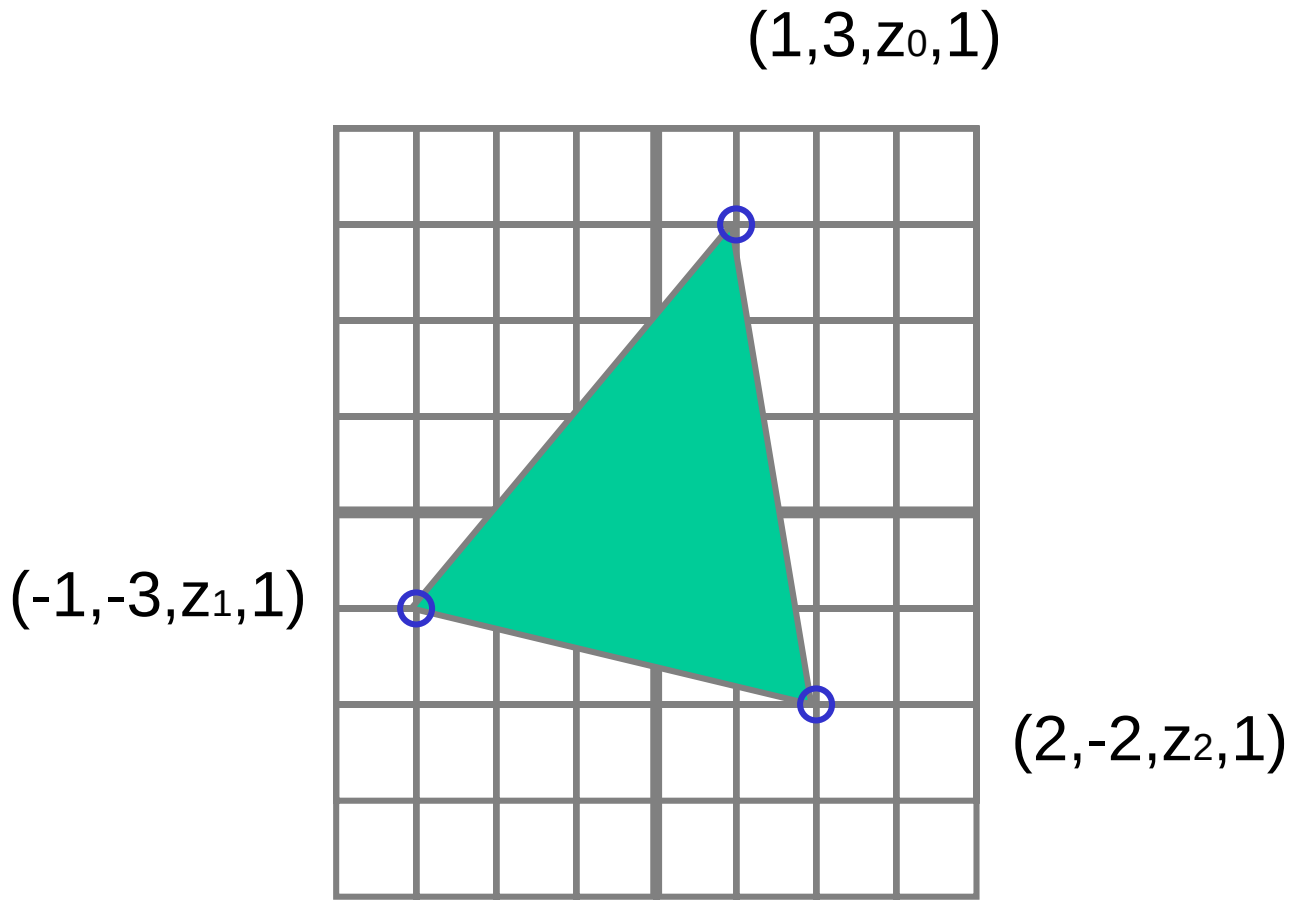
A New Robust Technique for Shadow Volume Capping

- Use stenciled shadow volume conventionally
  - Rather than using Carmack's Reverse
- Exploit rasterization with w=0 to draw semi-infinite polygons
  - Details follow
- Adjust depth range and projection matrix to ensure crack free near clip plane capping
  - Details follow
- Result: Robust stenciled shadow volume algorithm

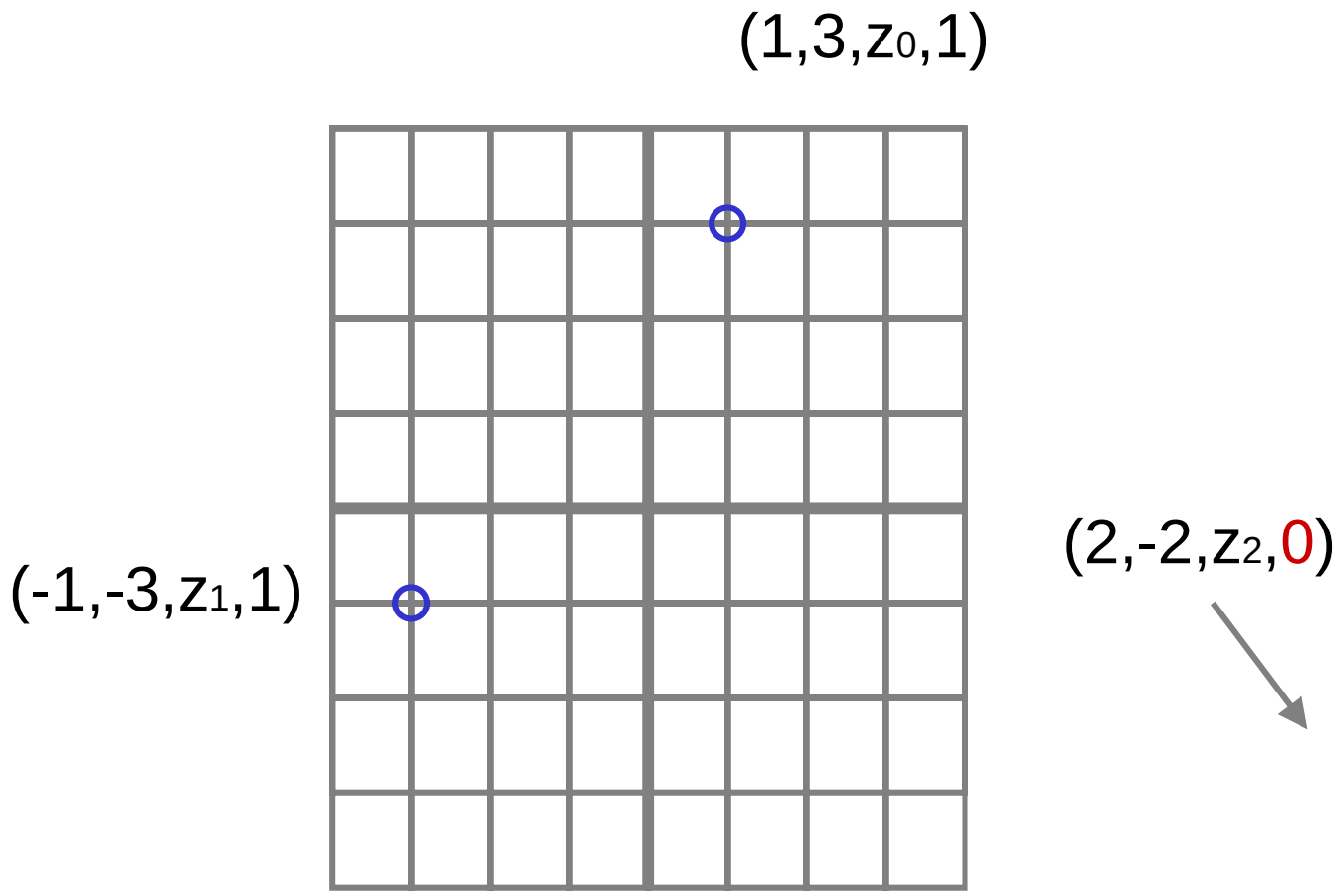
Rasterize This Polygon



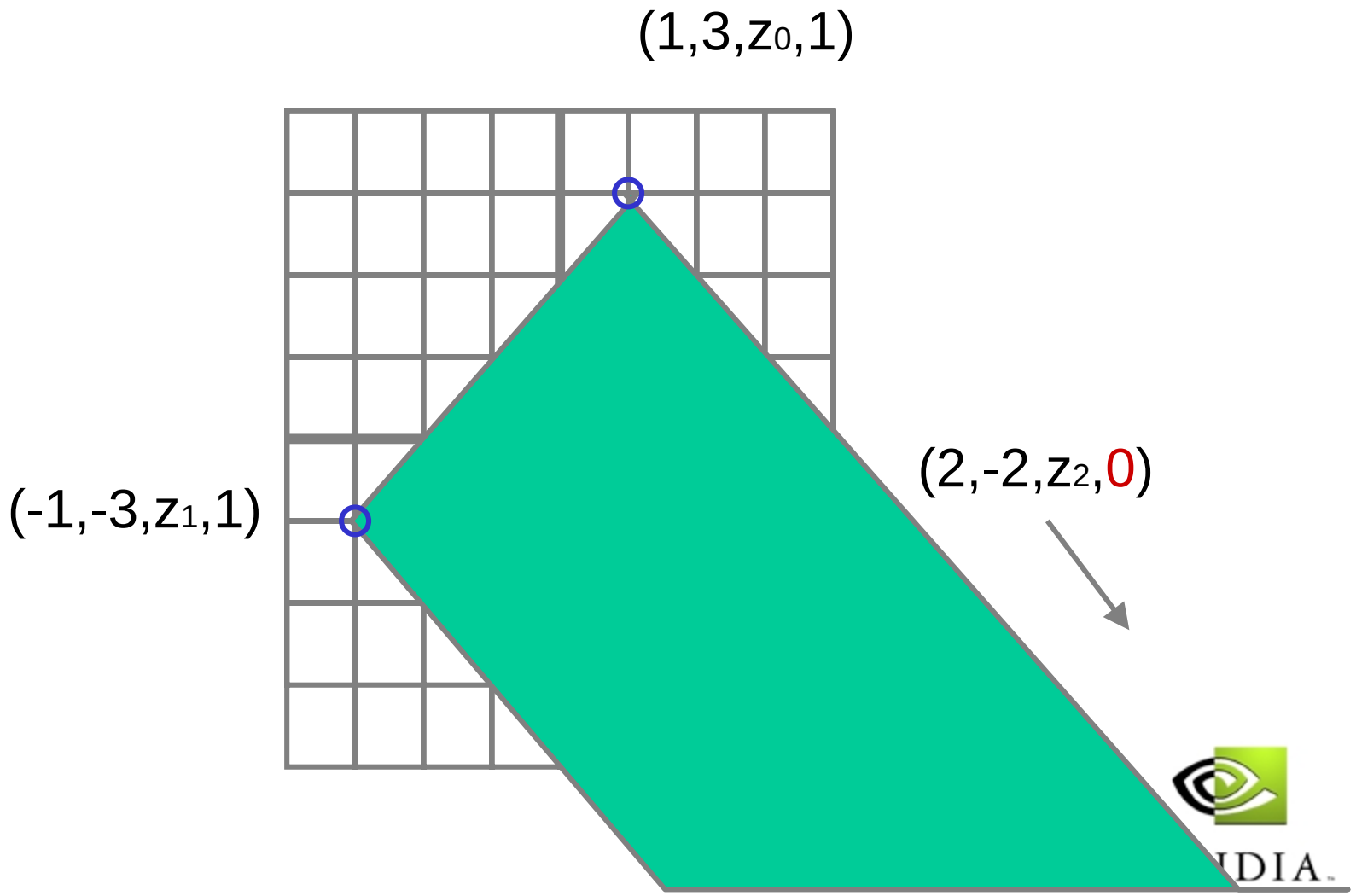
Rasterize This Polygon



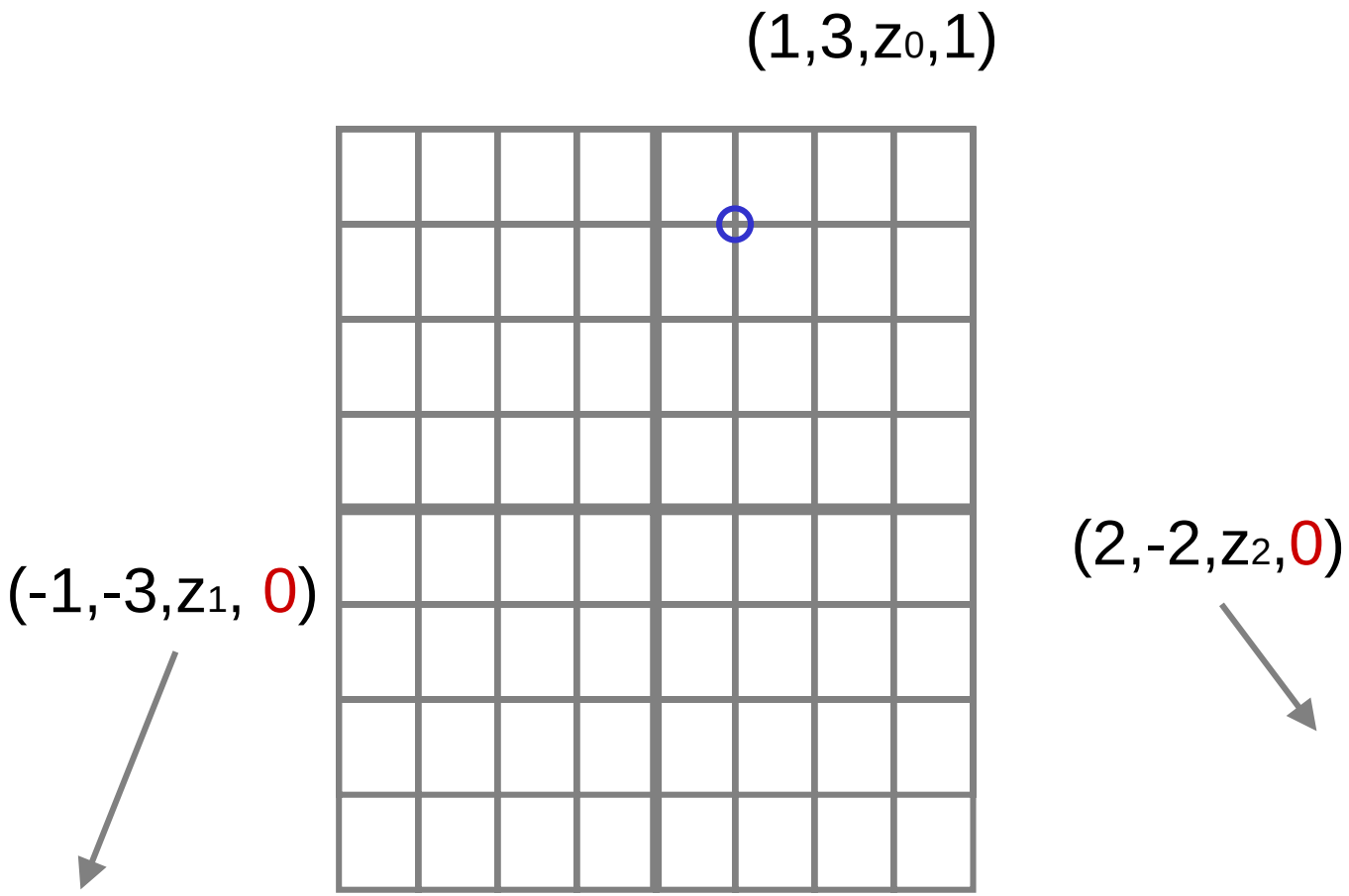
Rasterize This Polygon, One Vertex has W=0



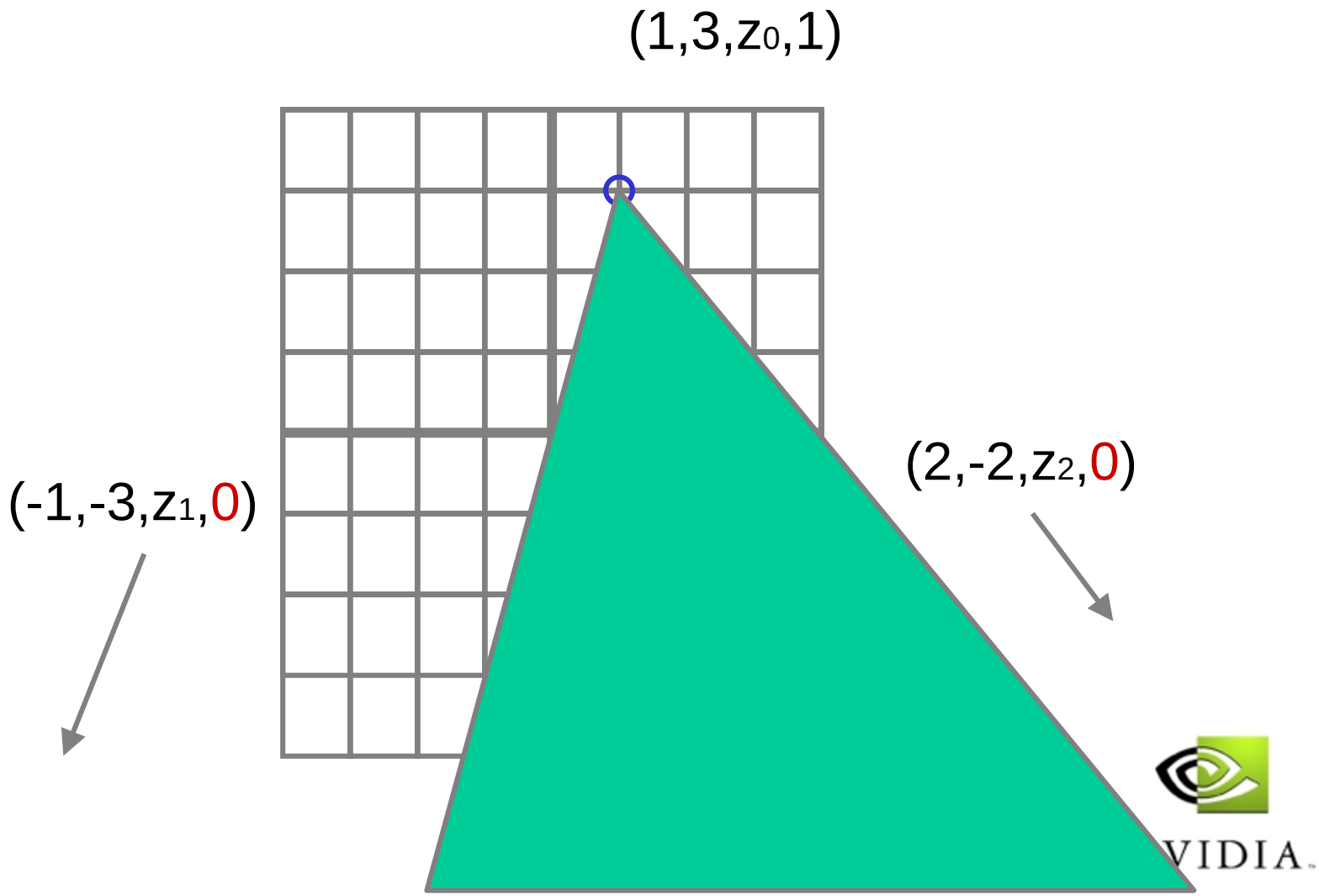
Rasterize This Polygon, One Vertex has W=0



Rasterize This Polygon



Rasterize This Polygon, Two Vertices have W=0



OpenGL Renders W=0 Polygons Correctly

- OpenGL implementations are required to render triangles properly when vertices have w=0
- Because of how the “possible silhouettes” for shadow volumes project on the near clip plane, rendering triangles with w=0 provides correct capping
  - Because we really need to be rendering semi-infinite polygons
  - That’s what w=0 vertices are all about

Near Clip Plane Capping Cases

- A “possible silhouette” is a loop of vertices on an occluding model
- Given a light position, this loop can be projected onto the near clip plane
  - We are only interested in projecting AWAY from the light (shadows just project away from a light)
- Three possibilities
  1. No vertices on the loop project away to the near clip plane (trivial case, no capping required)
  2. All vertices on the loop project away to the near clip plane (easy capping case)
  3. Some but not all of the loop vertices project away onto the near clip plane (hard capping case)

Handling the "Trivial" None Near Clip Plane Capping Case

- None of the rays cast away from the light source intersect with the near clip plane
- Render a quad strip loop using "possible silhouette" vertices and infinite vertices (with w=0) where x, y, and z are the direction away from the light
  - The w=0 vertices are easier to compute than trying to extend the shadow volume some sufficiently large finite distance
  - Avoid all worries that the large finite distance used may not actually be sufficient

Handling the "Easy" Near Clip Plane Capping Case

- Intersect each ray cast away from the light source with the near clip plane
- Render a quad strip loop using "possible silhouette" vertices and the near clip plane intersection positions
- Project all the model's triangles that are within the "possible silhouette" loop (i.e., facing away from the light) to the near clip plane and render them

Handling the "Hard" Near Clip Plane Capping Case (1)

- In this case, the "possible silhouette" loop projects AWAY onto the near clip plane for some loop vertices, but not all
- Render a quad strip loop using "possible silhouette" vertices and a projected position
  - If the ray cast from a loop vertex away from the light intersects the near clip plane, use the intersection position for the projected position
    - Send two vertices for the quad strip loop: the loop vertex and its projected position
  - Otherwise, …
    - Send two vertices for the quad strip loop: the loop vertex and its direction away from the light for XYZ and with w=0

Handling the "Hard" Near Clip Plane Capping Case (2)

- For each triangle within the "possible silhouette" loop (i.e., facing away from the light) …
  - If all three vertices do not project AWAY to the near clip plane, skip this triangle
  - If all three vertices do project AWAY to the near clip plane, render a triangle from these projected vertices
  - If one or two vertices project AWAY to the near clip plane, but not all three, these are handled as described on the next two slides

Handling the "Hard" Near Clip Plane Capping Case (3)

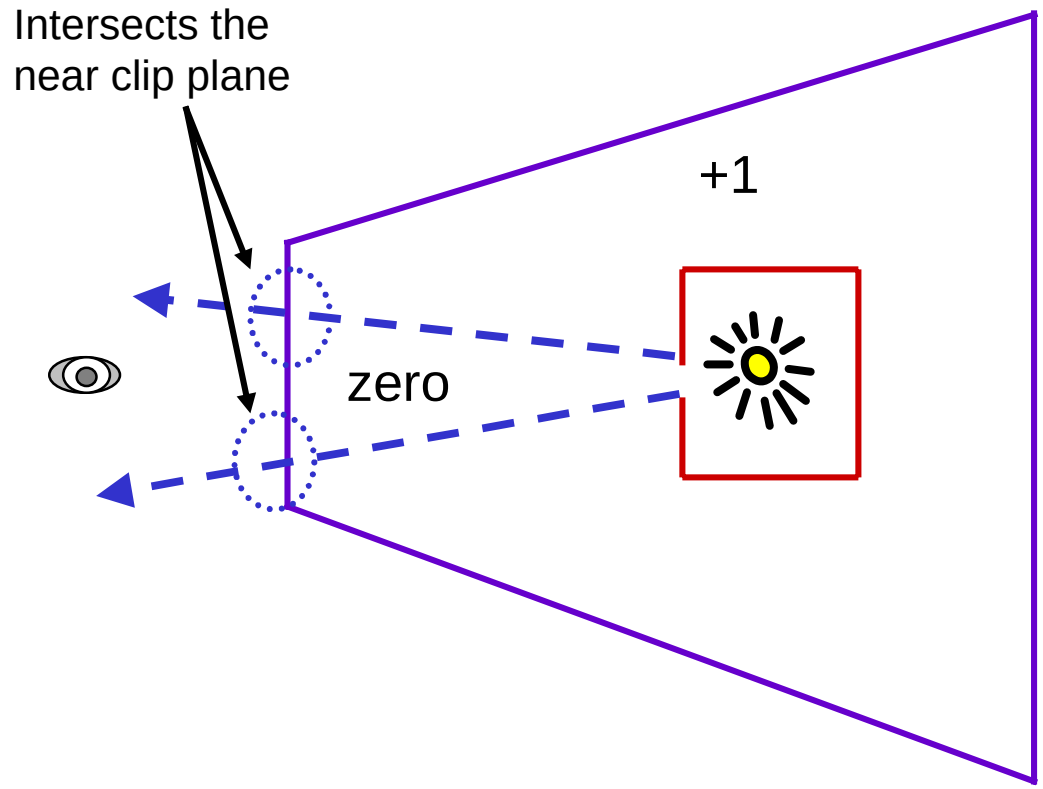
- If exactly one vertex projects AWAY to the near clip plane, render a triangle with this projected vertex and two vertices with w=0 as described
  - The two remaining vertices are determined by computing the plane containing the light position, the single projected vertex position on the near clip plane, and one of the two remaining loop vertices
  - Then take the cross product of the direction of this plane and the near clip plane
  - Render a triangle with the single projected vertex position and two other vertices such that the XYZ of these two vertices is the direction vector result from the cross product and w=0

Handling the "Hard" Near Clip Plane Capping Case (4)

- If exactly two vertices project AWAY to the near clip plane, render a quad with these two projected vertices and two other vertices with w=0 as described
  - Determine the third vertex by computing the plane containing the light position, one of the two projected vertices, and the unprojected loop vertex
  - Then take the cross product of the direction of this plane and the near clip plane
  - The third vertex is such that XYZ is the direction vector results from the cross product and w=0
  - Compute the fourth vertex in the same manner using the remaining projected vertex

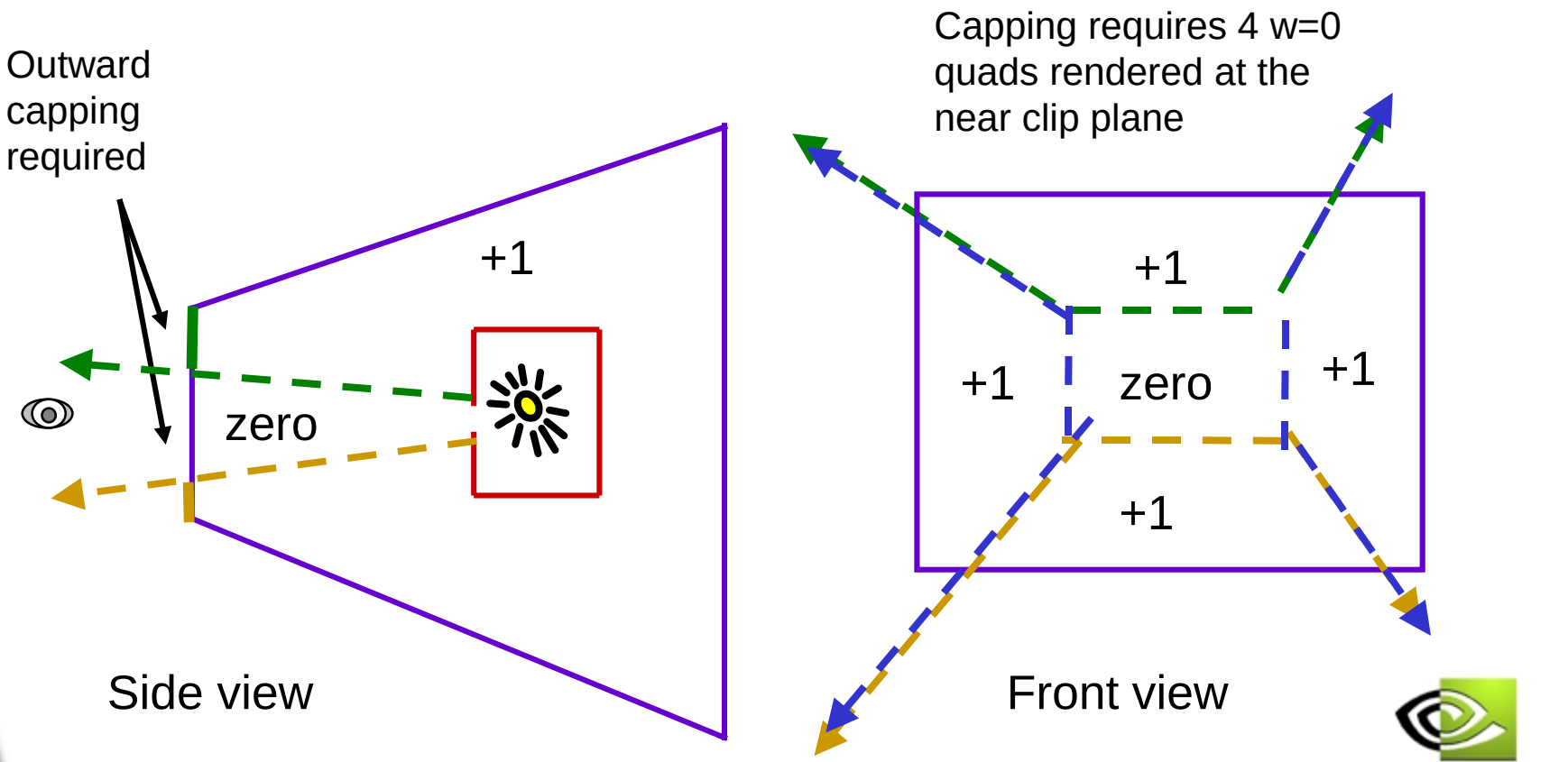
“Hard” Case Example

- **Consider a point light source in a box open on one side, facing the viewer (a.k.a. the lantern case)**



“Hard” Case Example

- **Lantern case requires w=0 capping**



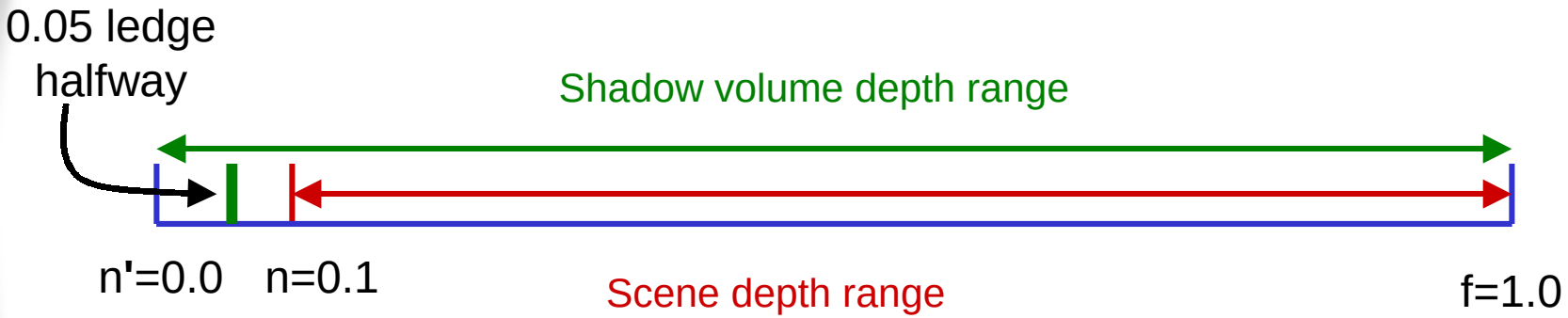
NVIDIA

Remaining Issues

- We must be extra careful to not introduce cracks when capping a stencil shadow volume at the near clip plane
  - We expect to draw the model in object space so would naturally render the shadow volume and cap it in object space too
  - The near clip plane capping must not introduce any T-junctions
- So far, we say that we are drawing the polygons at the near clip plane to cap our shadow volumes
  - However, the near clip plane is a razor's edge where polygons exactly or nearly coincident with the near clip plane may indeed be clipped

Building a “Ledge” at the Front of the Depth Buffer

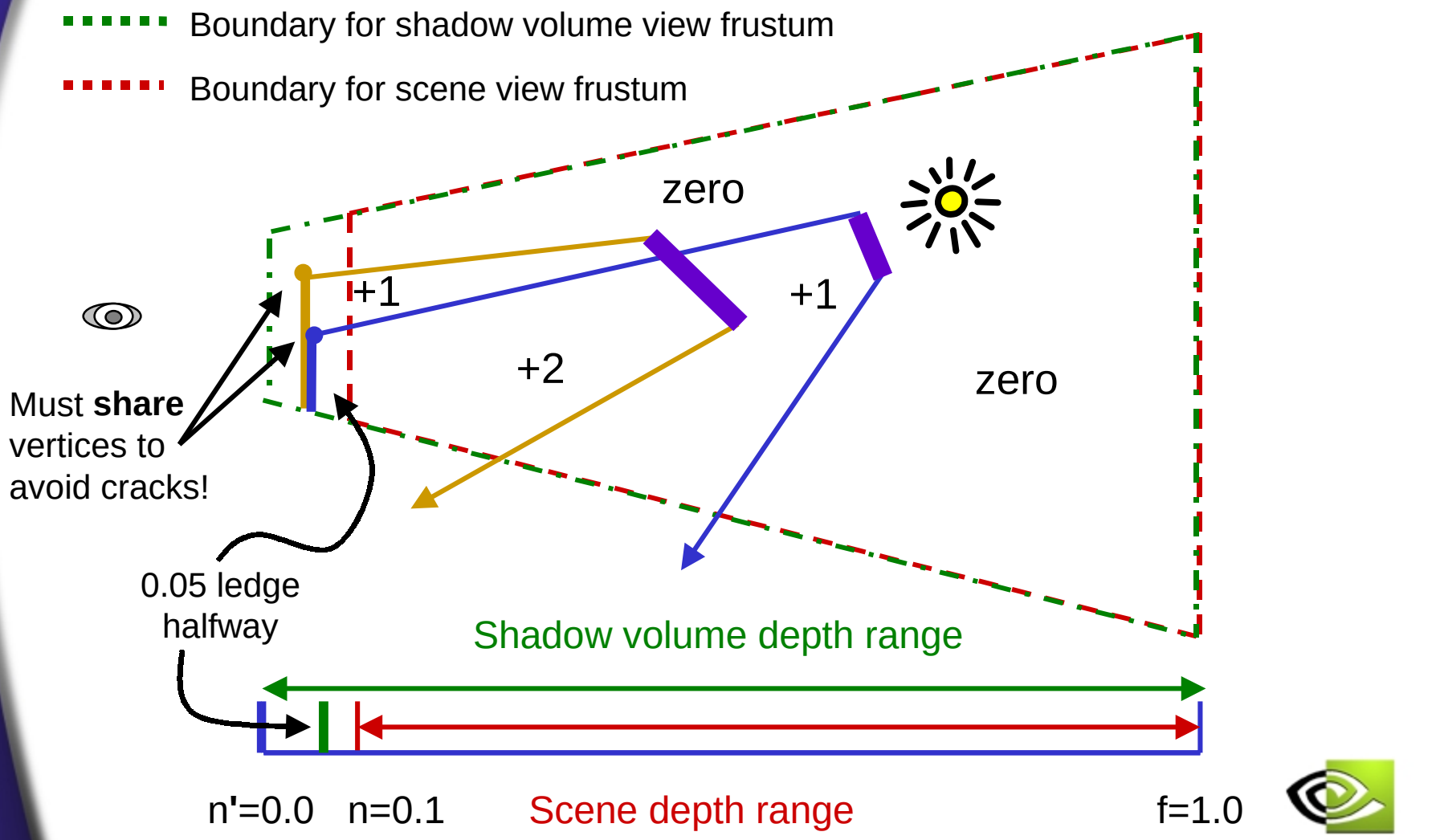
- **We would like to configure the depth buffer and projection matrix as follows:**



- **The scene would be drawn in the range [0.1 .. 1.0]**
- **The shadow volumes would be drawn in the range [0.0 .. 1.0]**
- **Shadow volume capping would be drawn at 0.05, the ledge halfway plane**
- **Depth values from the scene and shadow volumes must be comparable**

NVIDIA

Shared Vertices for Shadow Volume Capping on Ledge



NVIDIA

Math for Building a "Ledge" At Front of Depth Buffer (1)

- **Setting depth range and projection matrix**

  glDepthRange(f, n);
  gluPerspective(fov, aspect, N, F);

- **gluPerspective builds 4x4 matrix**

$$\begin{bmatrix} fov/aspect & 0 & 0 & 0 \\ 0 & fov & 0 & 0 \\ 0 & 0 & (F+N)/(N-F) & (2*F*N)/(N-F) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note: The Z row and only the Z row depends on F and N

X and Y positions can & will be bit exact if only Z row changes

NVIDIA

Math for Building a "Ledge" At Front of Depth Buffer (2)

- Given n, f, N, and F, and a closer depth range near value n', can we determine a value N' for the gluPerspective call that will make the depth values using n, f, N, and F comparable to the depth values generated using n', f, N', and F ?
- Being comparable mathematically means:

$$\frac{(f-n)}{2} * \frac{(FN)/(N-F)\ z + (2*F*N)/(N-F)}{-z} + \frac{(n+f)}{2}$$

$$=$$

$$\frac{(f-n')}{2} * \frac{(FN')/(N'-F)\ z + (2*F*N')/(N'-F)}{-z} + \frac{(n'+f)}{2}$$

NVIDIA

Math for Building a "Ledge" At Front of Depth Buffer (3)

- The expression

$$\frac{(f-n)}{2} * \frac{(FN)/(N-F)\ z + (2*F*N)/(N-F)}{-z} + \frac{(n+f)}{2}$$

$$=$$

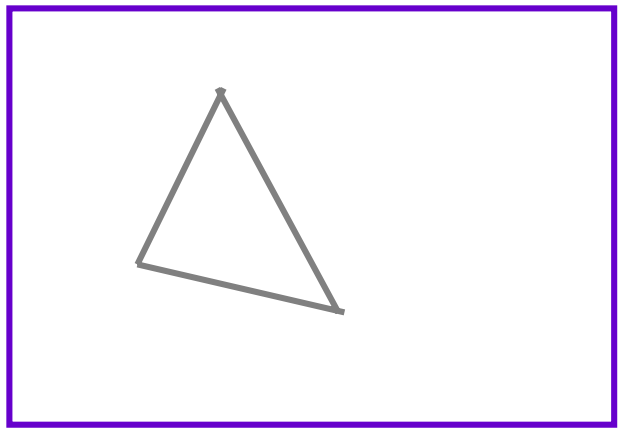
$$\frac{(f-n')}{2} * \frac{(FN')/(N'-F)\ z + (2*F*N')/(N'-F)}{-z} + \frac{(n'+f)}{2}$$

does simplify so that z cancels out of the expression. This means z is comparable. Moreover, N' can be expressed in terms of n, f, N, F, and n':

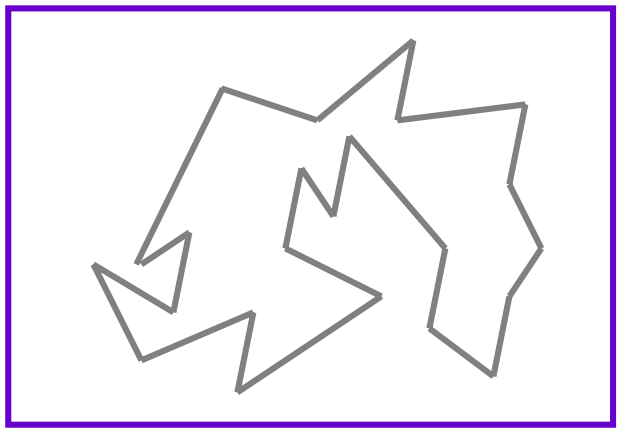
$$N' = \frac{-(f-n)*F*N}{n*N - f*F - n'*(F-N)}$$

Issue: How to Properly Tessellate Capping Polygons

- **Rendering polygons at near clip plane properly caps shadow volumes**
  - **But proper capping is hard if you *only* assume that you know the outline**



Trivial to cap the outline of a single triangle



Non-trivial if just given a complex shadow volume intersection with the near clip plane

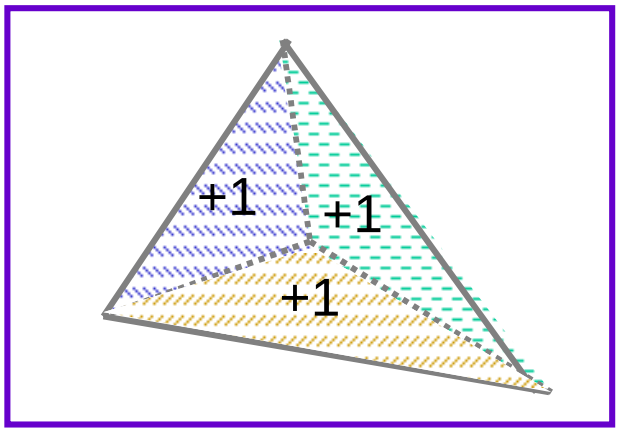
NVIDIA.

Solution: Project Object Polygons to Near Plane

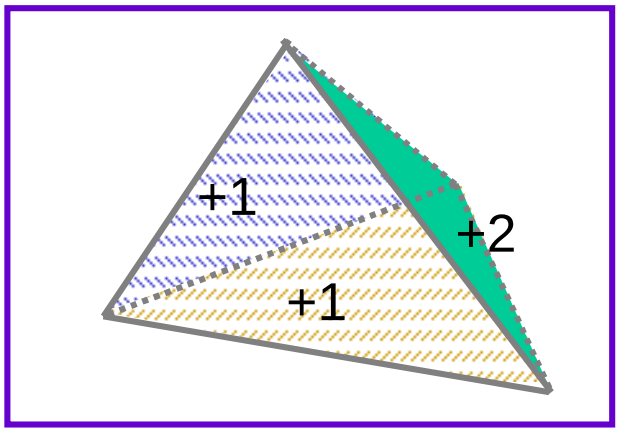
- Tessellating from an outline is a hard problem
  - Involved computational geometry problem
- But we have more information than the outline
  - Capping tessellation on the near clip plane is really just a projection of object's back facing geometry (with respect to the light) onto the near clip plane
  - So render the capping geometry on the near clip plane by projection object's back facing geometry to the near clip plane

Incrementing for Projected Polygons Causes Errors

- **Naively, increment stencil for projected capping polygons**
- **Consider the following two situations**



Good Case:
Incrementing works correctly, all pixels inside outline are incremented



Bad Case: But when interior vertex is outside of outline, incorrect double incrementing occurs outside the outline!

When Does the Bad Case Occur

- Numerical error in transformation can lead to interior vertices that are barely outside of the outline when projected to screen space
  - Trust me – it happens
- If not corrected, double incrementing of regions that should not be incremented at all leads to spurious shadowed pixels
  - Only a few pixels, but obvious
  - Particularly obvious when animating

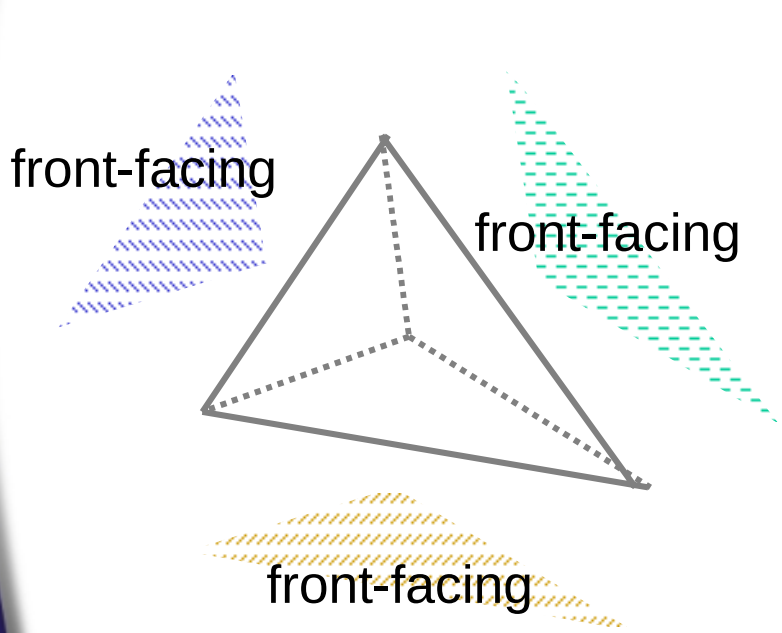
Example of Bad Case

- **Bogus Shadowed polygonal region due to naïve capping**

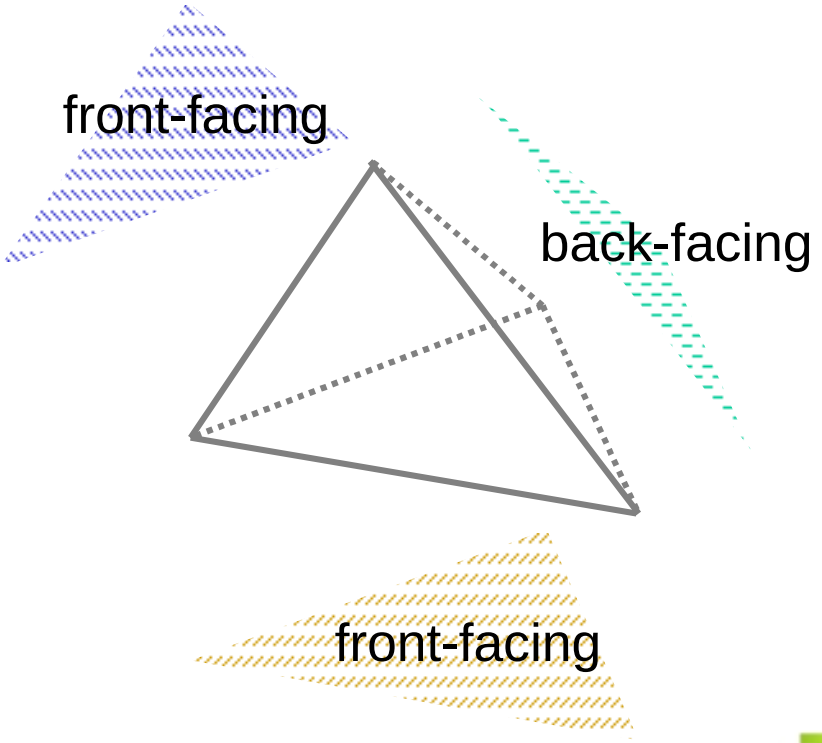
Notice how the bogus shadowed polygon is near the capping outline

Examine the Bad Case in More Detail

- **Examine polygon facing-ness for the two example situations**



Good Case

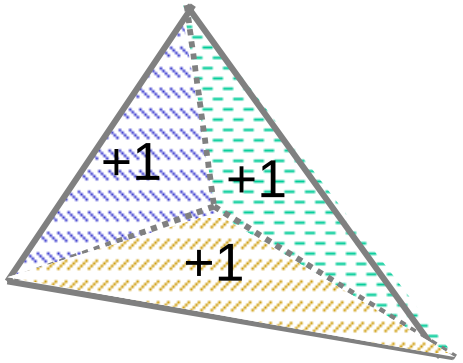


Bad Case

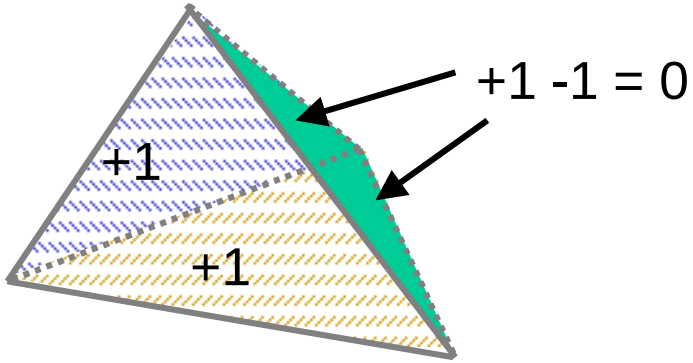
NVIDIA

Solution: Increment Front Faces, Decrement Back Faces

- **Draw capping polygons twice**
  - **Increment stencil for front faces**
  - **Decrement stencil for back faces**



All faces front facing



Region that was double incremented is now left zero because it is both incremented & decremented

NVIDIA

Performance Implications

- Is drawing capping polygons twice expensive?
- Not really
  - Most capping polygons are front-facing
  - Rarely are back-facing polygons encountered
  - When back-facing polygons occur, there is an extra overhead of incrementing and decrementing pixels (pixel touched twice for no net effect)
  - Modern hardware culls wrong-facing polygons at amazing rates
    - Hardware setup can cull 25+ million wrong-facing polygons per second
  - Moreover, the alternative is bogus shadowing

High-level Shadow Volume Performance Optimization

- Key performance problem with shadow volumes
  - shadow volumes consume *invisible* fill rate
- If you plan to use shadow volumes
  - Invest early on in developing a robust algorithm
  - Analyze your maps to eliminate shadow volumes that cannot be seen
    - Example: shadows completely behind walls
  - Do not overextend shadow volumes
    - Example: do not extend shadow volumes past walls
- Carmack's optimizations in these areas will make new Doom engine awesome

NVIDIA.

Combining Shadow Volumes With Bump Mapped Models

md2shader demo/example