



# Game Math Case Studies

**Eric Lengyel, PhD**  
Terathon Software

GAME DEVELOPERS CONFERENCE®  
MOSCONC CENTER · SAN FRANCISCO, CA  
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015



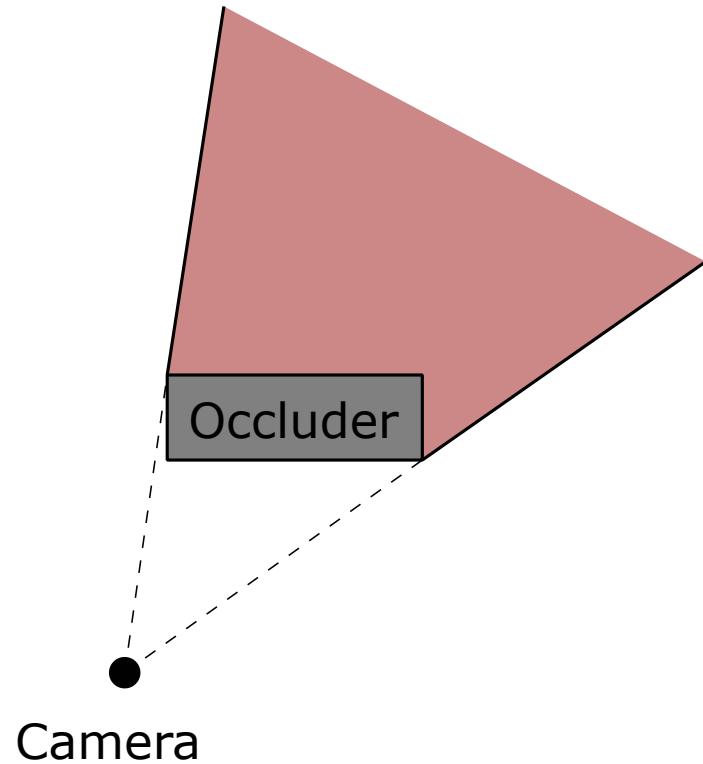
# Content of this Talk

- Real-world problems from game dev
  - Small problems, that is, and easy to state
- Actual solutions used in shipping games
  - Using math that's not too advanced
- Strategies for finding elegant solutions



# Occlusion boxes

- Plain boxes put in world as occluders
- Extrude away from camera to form occluded region of space where objects don't need to be rendered
- How to do this most efficiently?





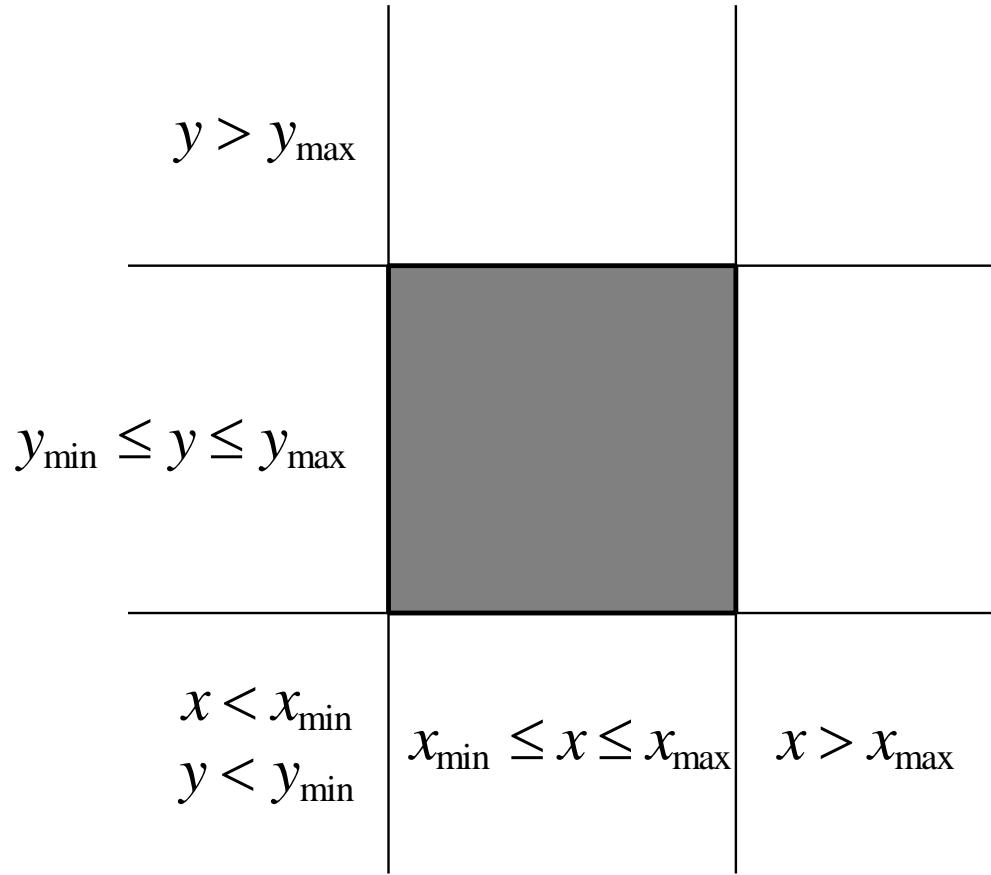
# Occlusion boxes

- Could classify box faces as front/back and find silhouette edges
  - Similar to stencil shadow technique
- A better solution accounts for small solution space



# Occlusion boxes

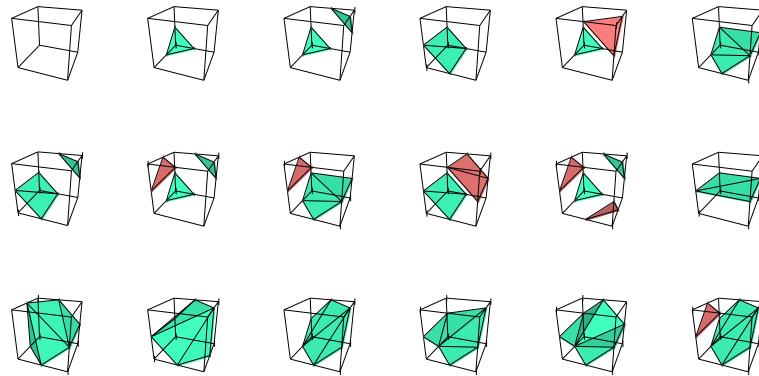
- There are exactly 26 possible silhouettes
- Three possible states for camera position on three different axes
  - $\text{position} < \text{box min}$
  - $\text{position} > \text{box max}$
  - $\text{box min} \leq \text{position} \leq \text{box max}$
  - Inside box excluded



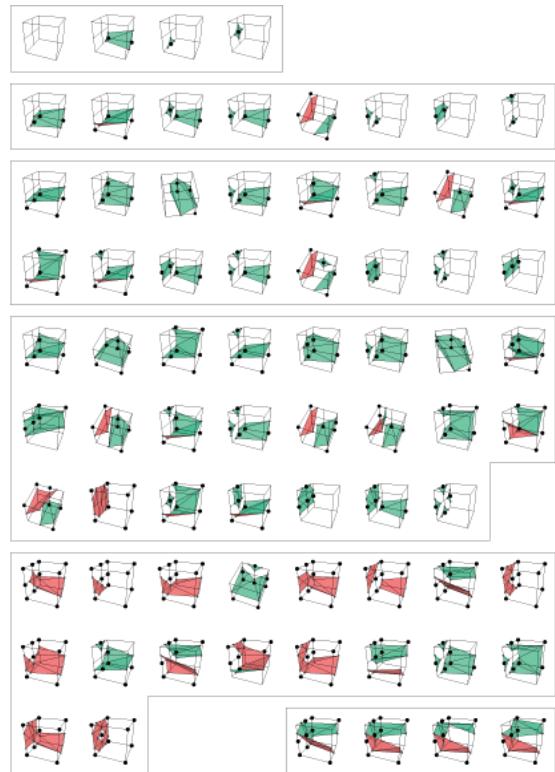
condition	code
$x > x_{\max}$	0x01
$x < x_{\min}$	0x02
$y > y_{\max}$	0x04
$y < y_{\min}$	0x08
$z > z_{\max}$	0x10
$z < z_{\min}$	0x20



# Finite classifications



**Marching Cubes, fixed polarity**  
(256 cases, 18 classes)



**Transvoxel Algorithm**  
(512 cases, 73 classes)

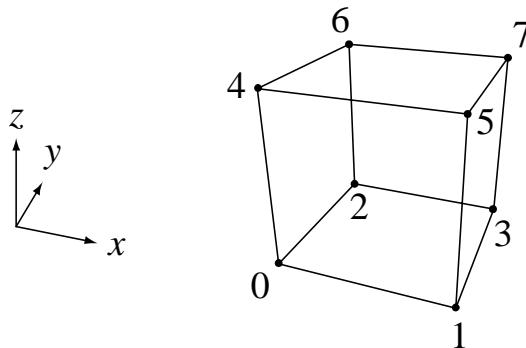


# Occlusion boxes

- Calculate camera position state and use table to get silhouette
- Always a closed convex polygon with exactly 4 or 6 vertices and edges



# Occlusion boxes



```
// Upper 3 bits = vertex count, lower 5 bits = polygon index
const unsigned_int8 occlusionPolygonIndex[43] =
{
    0x00, 0x80, 0x81, 0x00, 0x82, 0xC9, 0xC8, 0x00, 0x83, 0xC7, 0xC6, 0x00, 0x00, 0x00, 0x00,
    0x84, 0xCF, 0xCE, 0x00, 0xD1, 0xD9, 0xD8, 0x00, 0xD0, 0xD7, 0xD6, 0x00, 0x00, 0x00, 0x00,
    0x85, 0xCB, 0xCA, 0x00, 0xCD, 0xD5, 0xD4, 0x00, 0xCC, 0xD3, 0xD2
};

// All 26 polygons with vertex indexes from diagram on left
const unsigned_int8 occlusionVertexIndex[26][6] =
{
    {1, 3, 7, 5},
    {2, 0, 4, 6},
    {3, 2, 6, 7},
    {0, 1, 5, 4},
    {4, 5, 7, 6},
    {1, 0, 2, 3},
    {2, 0, 1, 5, 4, 6},
    {0, 1, 3, 7, 5, 4},
    {3, 2, 0, 4, 6, 7},
    {1, 3, 2, 6, 7, 5},
    {1, 0, 4, 6, 2, 3},
    {5, 1, 0, 2, 3, 7},
    {4, 0, 2, 3, 1, 5},
    {0, 2, 6, 7, 3, 1},
    {0, 4, 5, 7, 6, 2},
    {4, 5, 1, 3, 7, 6},
    {1, 5, 7, 6, 4, 0},
    {5, 7, 3, 2, 6, 4},
    {3, 1, 5, 4, 6, 2},
    {2, 3, 7, 5, 4, 0},
    {1, 0, 4, 6, 7, 3},
    {0, 2, 6, 7, 5, 1},
    {7, 6, 2, 0, 1, 5},
    {6, 4, 0, 1, 3, 7},
    {5, 7, 3, 2, 0, 4},
    {4, 5, 1, 3, 2, 6}
};
```



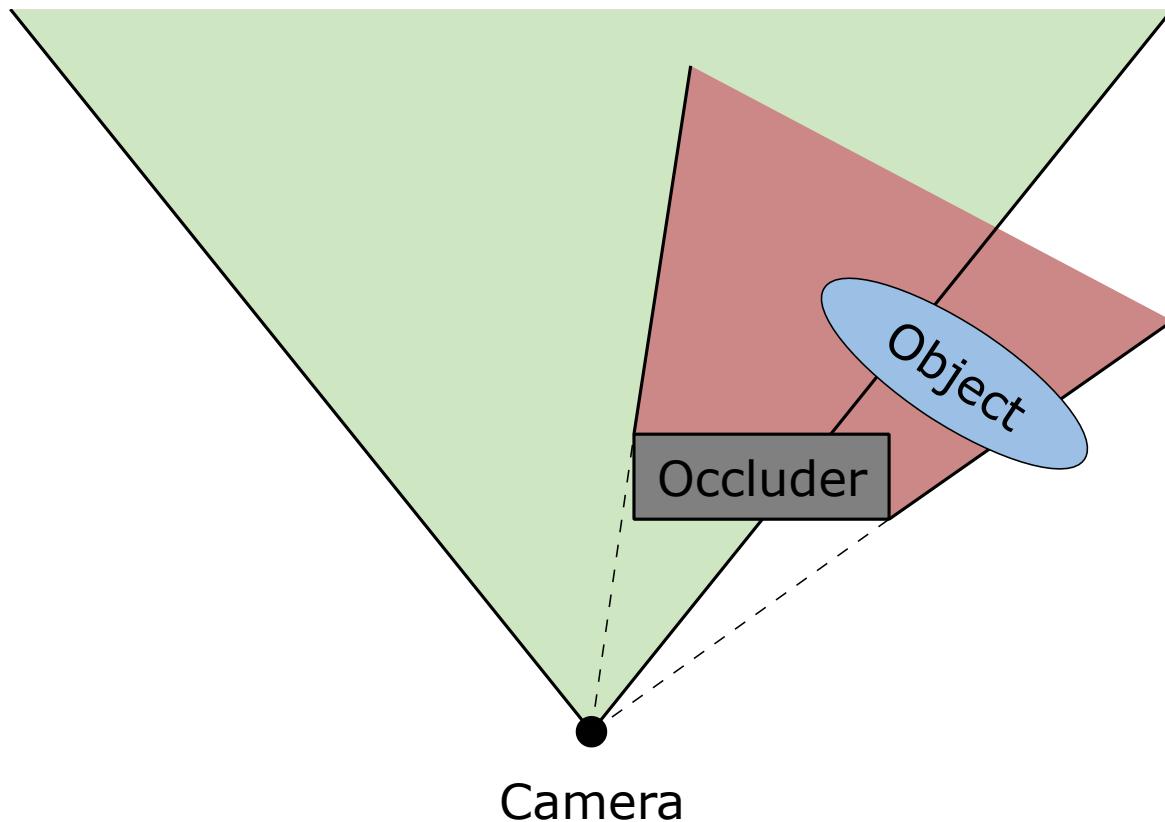
# Occlusion boxes

- Any silhouette edge that is off screen can be eliminated to make occlusion region larger
- Gives occluder infinite extent in that direction
- Allows more objects to be occluded because they must be completely inside extruded silhouette to be hidden



# Occlusion boxes

- Silhouette edge is culled if both vertices on negative side of some frustum plane
- *And* extruded plane normal and frustum plane normal have positive dot product





# Occlusion boxes

- Strategy:

Look for ways to classify solutions

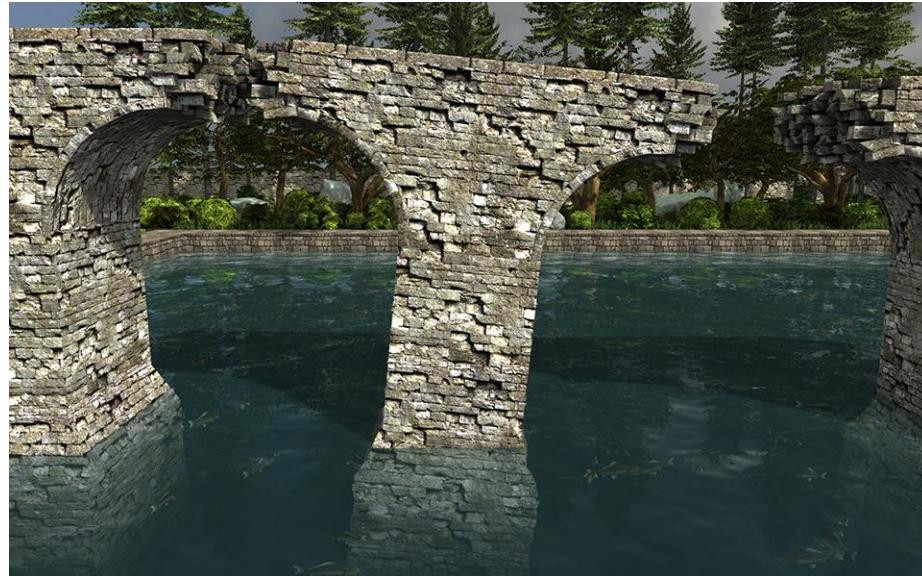


# Oblique near plane trick

- Sometimes need a clipping plane for a flat surface in scene
- For example, water or mirror
  - Prevent submerged objects from appearing in reflection



Ordinary frustum



Oblique near plane



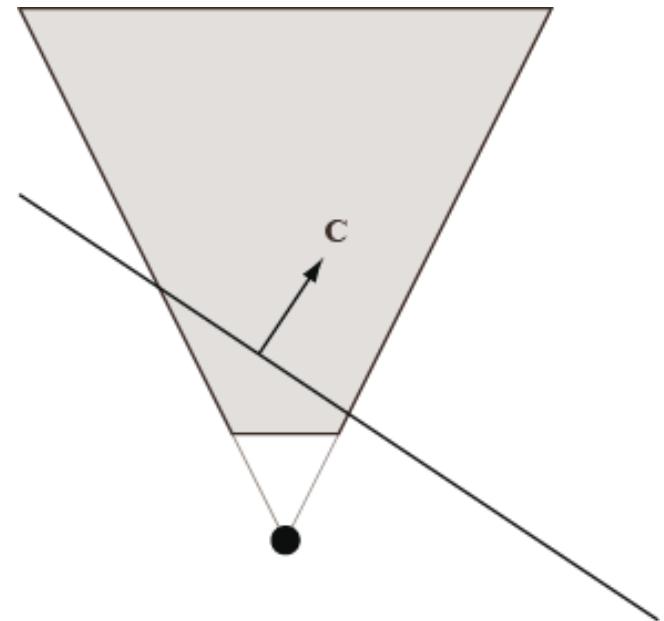
# Oblique near plane trick

- Hardware clipping plane?
  - May not even be supported
  - Requires shader modification
  - Could be slower



# Oblique near plane trick

- Extra clipping plane almost always redundant with near plane
- Don't need to clip to both





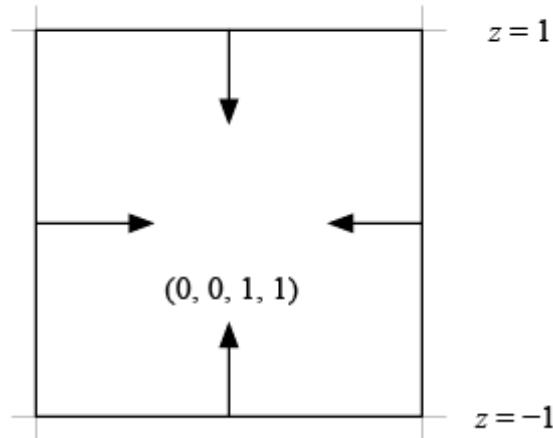
# Oblique near plane trick

- Possible to modify projection matrix
- Move near plane to arbitrary location
- No extra clipping plane, no redundancy



# Oblique near plane trick

- In normalized device coordinates (NDC),  
near plane has coordinates  $(0,0,1,1)$





# Oblique near plane trick

- Planes (row antivectors) are transformed from NDC to camera space by right multiplication by the projection matrix
- So the plane  $(0, 0, 1, 1)$  becomes  $\mathbf{M}_3 + \mathbf{M}_4$ , where  $\mathbf{M}_i$  is the  $i$ -th row of the projection matrix



# Oblique near plane trick

- $\mathbf{M}_4$  must remain  $(0, 0, -1, 0)$  so that perspective correction still works right
- Let  $\mathbf{C} = (C_x, C_y, C_z, C_w)$  be the camera-space plane that we want to clip against
  - Assume  $C_w < 0$ , camera on negative side
- We must have  $\mathbf{C} = \mathbf{M}_3 + (0, 0, -1, 0)$



# Oblique near plane trick

- $\mathbf{M}_3 = \mathbf{C} - \mathbf{M}_4 = (C_x, C_y, C_z + 1, C_w)$

$$\mathbf{M} = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ C_x & C_y & C_z + 1 & C_w \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- This matrix maps points on the plane  $\mathbf{C}$  to the plane  $z = -1$  in NDC

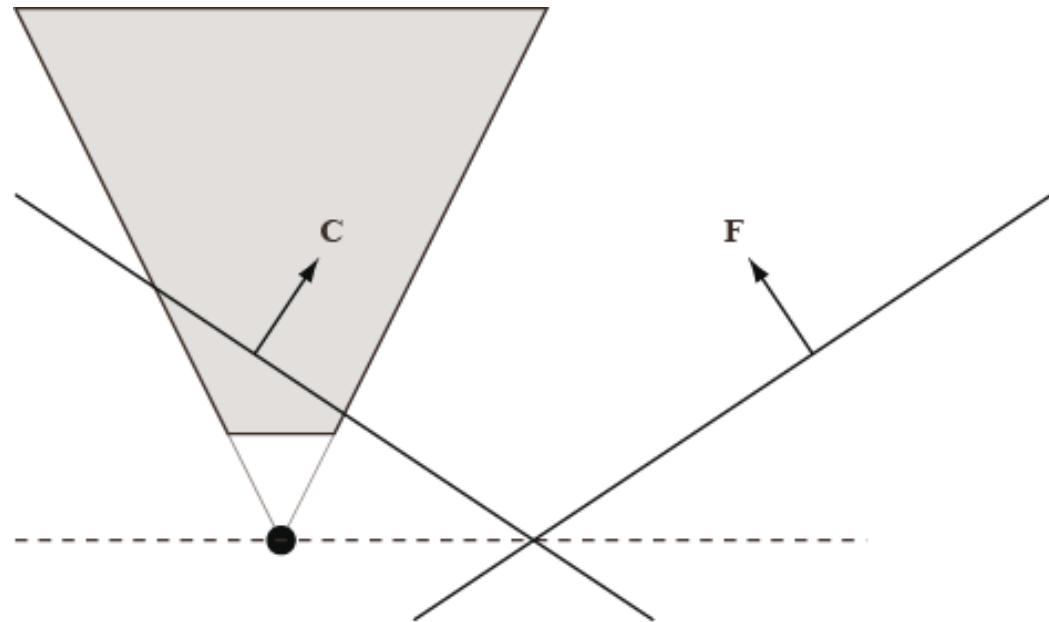


# Oblique near plane trick

- But what happens to the far plane?
- $F = M_4 - M_3 = 2M_4 - C$
- Near plane and (negative) far plane differ only in the  $z$  coordinate
- Thus, they must coincide where they intersect the  $z = 0$  plane



# Oblique near plane trick





# Oblique near plane trick

- Far plane is a complete mess
- Depths in NDC no longer represent distance from camera plane, but correspond to some skewed direction between near and far planes
- We can minimize the effect,  
and in practice it's not so bad



# Oblique near plane trick

- We still have a free parameter:  
the clipping plane **C** can be scaled
- Scaling **C** has the effect of changing the orientation of the far plane **F**
- We want to make the new view frustum as small as possible while still including the conventional view frustum



# Oblique near plane trick

- Let  $\mathbf{F} = 2\mathbf{M}_4 - a\mathbf{C}$
- Choose the point  $\mathbf{Q}$  which lies furthest opposite the near plane in NDC:

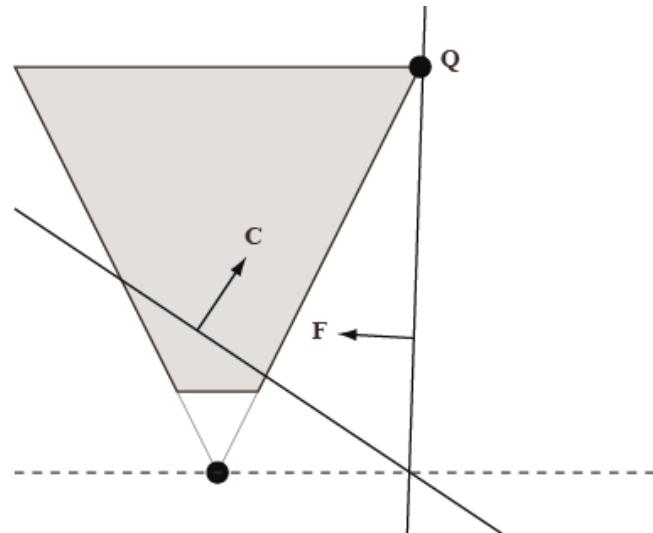
$$\mathbf{Q} = \mathbf{M}^{-1} \cdot (\text{sgn}(C_x), \text{sgn}(C_y), 1, 1)$$

- Solve for  $a$  such that  $\mathbf{Q}$  lies in plane  $\mathbf{F}$ : 
$$a = \frac{\mathbf{M}_4 \wedge \mathbf{Q}}{\mathbf{C} \wedge \mathbf{Q}}$$



# Oblique near plane trick

- Near plane doesn't move, but far plane becomes optimal





# Oblique near plane trick

- Works for any perspective projection matrix
  - Even with infinite far depth
- More analysis available in "[Oblique Depth Projection and View Frustum Clipping](#)", Journal of Game Development, Vol. 1, No. 2.



# Oblique near plane trick

- Strategy:

Get the big picture

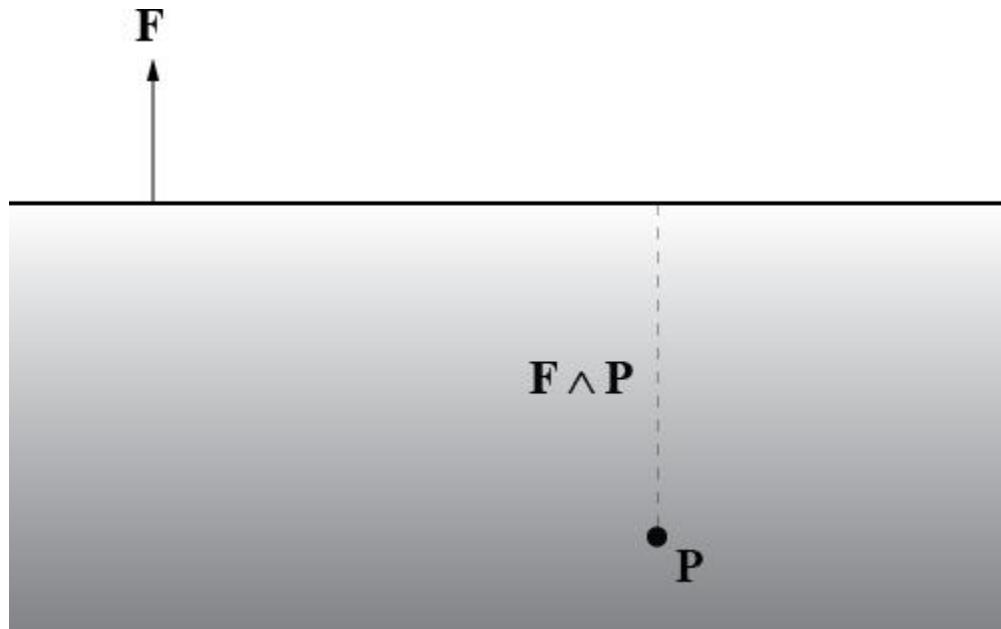


# Fog bank occlusion

- Consider fog bank bounded by plane
- Linear density gradient with increasing depth
  - Perpendicular to plane
  - Zero density at plane



# Fog bank occlusion





# Fog bank occlusion



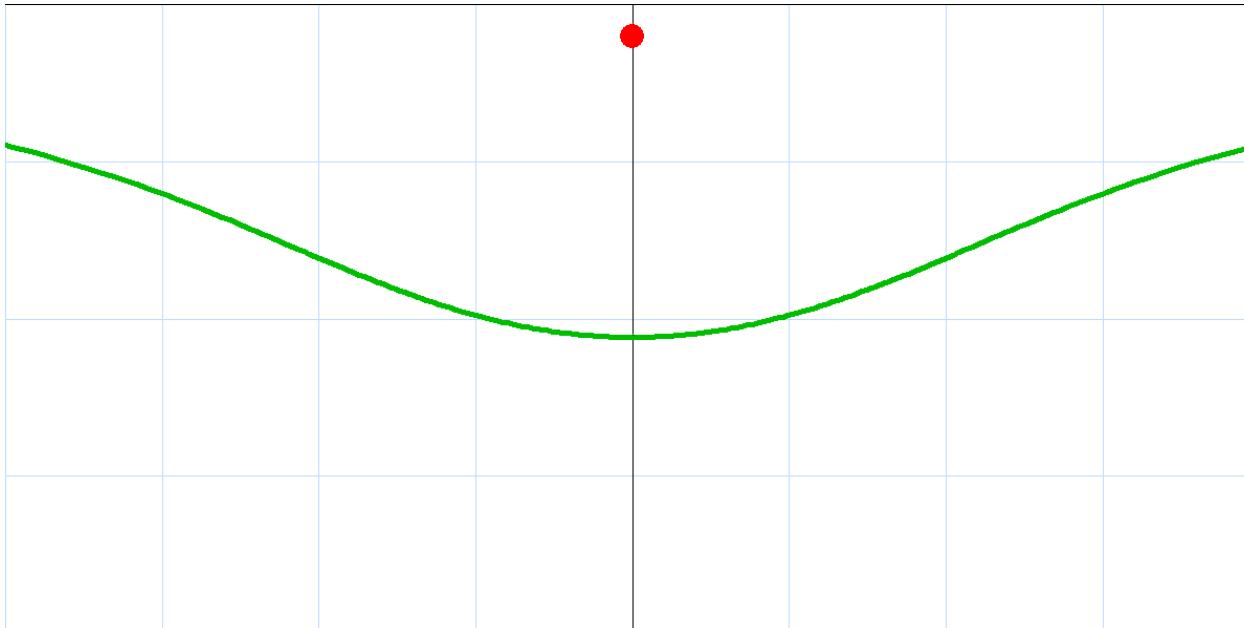


# Fog bank occlusion

- For a given camera position, we want to cull objects that are completely fogged
- Surface beyond which objects are completely fogged is interesting...



# Fog bank occlusion



(video)

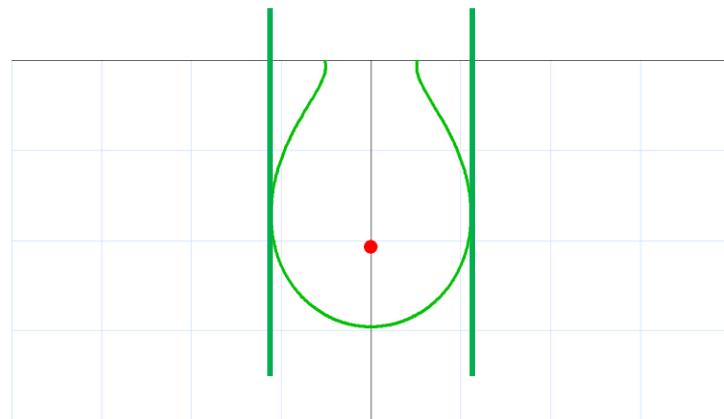
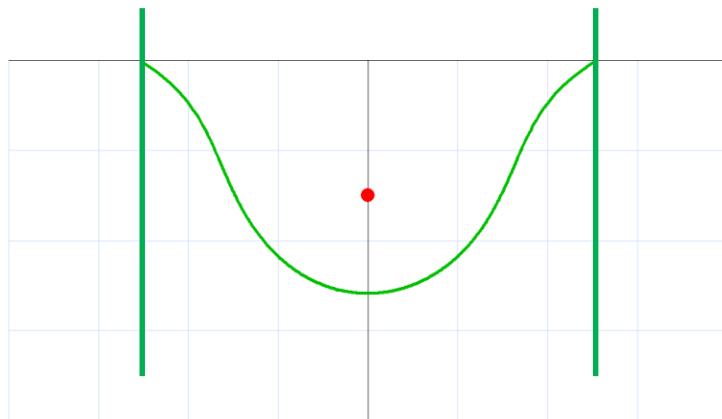


# Fog bank occlusion

- Culling against that curve is impractical
- Instead, calculate its maximum extent parallel to the fog plane
- Then cull against plane perpendicular to fog plane and camera view direction at that distance



# Fog bank occlusion





# Fog bank occlusion

- $\mathbf{F}$  = fog plane, normal outward
- $\mathbf{C}$  = camera position
- $\mathbf{P}$  = point being shaded
- $\mathbf{V} = \mathbf{C} - \mathbf{P}$
- $a$  = linear density coefficient



# Fog bank occlusion

- Density as function of depth:

$$\rho(\mathbf{P}) = -a(\mathbf{F} \wedge \mathbf{P})$$

- Log light fraction  $g(\mathbf{P})$  for given  $\mathbf{C}$  and  $\mathbf{P}$ :<sup>†</sup>

$$g(\mathbf{P}) = -a\|\mathbf{V}\| \frac{\mathbf{F} \wedge \mathbf{P} + \mathbf{F} \wedge \mathbf{C}}{2}$$

<sup>†</sup>See "[Unified Distance Formulas for Halfspace Fog](#)", *Journal of Graphics Tools*, Vol. 12, No. 2 (2007).



# Fog bank occlusion

- Set  $g(\mathbf{P})$  to log of small enough fraction to be considered fully fogged
- For example:  $g(\mathbf{P}) = -\log(1/256)$
- This is constant



# Fog bank occlusion

- For given **C**, we need to find **P** with the maximum horizontal distance *d* from **C** that also satisfies

$$g(\mathbf{P}) = -a \|\mathbf{V}\| \frac{\mathbf{F} \wedge \mathbf{P} + \mathbf{F} \wedge \mathbf{C}}{2}$$



# Fog bank occlusion

- So express  $d$  as a function of  $\mathbf{P}$  and find the zeros of the derivative, right?
- Turns out to be a huge mess
- Not clear that good solution exists



# Fog bank occlusion

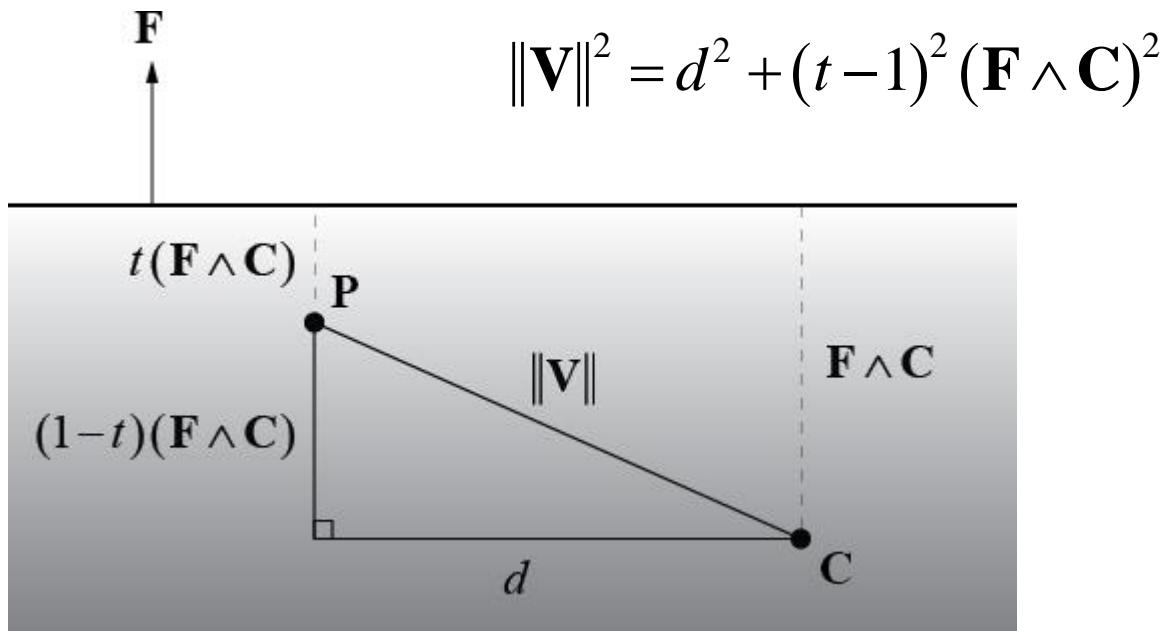
- Insight: instead of using independent variable  $\mathbf{P}$ , express  $\mathbf{F} \wedge \mathbf{P}$  as a fraction of  $\mathbf{F} \wedge \mathbf{C}$

$$\mathbf{F} \wedge \mathbf{P} = t(\mathbf{F} \wedge \mathbf{C})$$

$$g(\mathbf{P}) = -a\|\mathbf{V}\| \frac{(t+1)(\mathbf{F} \wedge \mathbf{C})}{2}$$



# Fog bank occlusion





# Fog bank occlusion

- Can now write  $g(\mathbf{P})$  as follows

$$g(\mathbf{P}) = -\frac{a}{2}(t+1)(\mathbf{F} \wedge \mathbf{C}) \sqrt{d^2 + (t-1)^2 (\mathbf{F} \wedge \mathbf{C})^2}$$

- And solve for  $d^2$ :

$$d^2 = \frac{m^2}{(t+1)^2 (\mathbf{F} \wedge \mathbf{C})^2} - (t-1)^2 (\mathbf{F} \wedge \mathbf{C})^2 \quad m = \frac{2g(\mathbf{P})}{a}$$



# Fog bank occlusion

- Take derivative, set to zero, simplify:

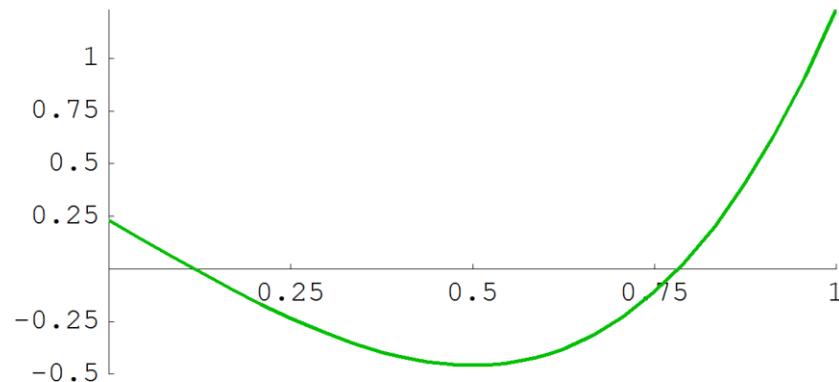
$$t^4 + 2t^3 - 2t + k - 1 = 0 \quad k = \frac{m^2}{(\mathbf{F} \wedge \mathbf{C})^4}$$

- Now need to solve quartic polynomial



# Fog bank occlusion

- Know what your functions look like!



- Always  $k$  at  $t = 1$ , local min at  $t = 1/2$



# Fog bank occlusion

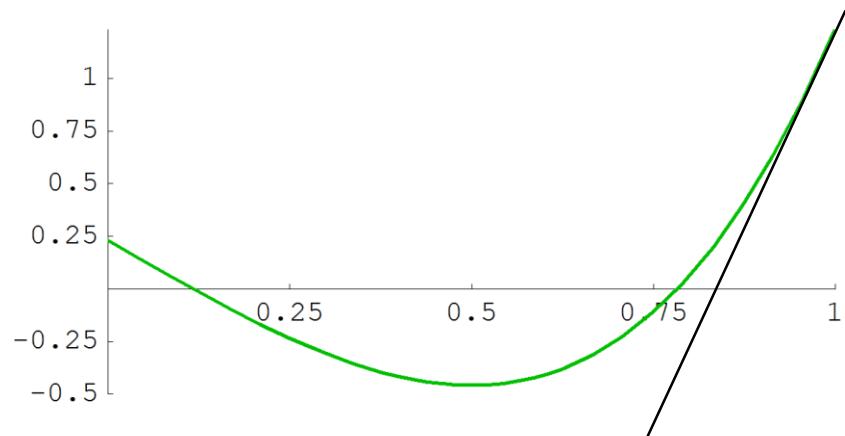
- If function is negative at  $t = 1/2$ ,  
then solution exists
  - Happens exactly when  $k < 27/16$
- Tempting to calculate with closed-form solution  
to quartic
- But almost always better to use  
Newton's method, especially in this case



# Fog bank occlusion

- Newton's method:

$$t_{i+1} = t_i - \frac{f(t)}{f'(t)}$$





# Fog bank occlusion

- In our case,

$$f'(t) = 4t^3 + 6t^2 - 2$$

- Start with  $t_0 = 1$ :

$$f(1) = k$$

$$f'(1) = 8$$



# Fog bank occlusion

- Calculate first iteration explicitly:

$$t_1 = 1 - \frac{k}{8}$$

- Newton's method converges very quickly with 1–2 more iterations



# Fog bank occlusion

- Plug  $t$  back into function for  $d^2$  to get culling distance

$$d^2 = \frac{m^2}{(t+1)^2 (\mathbf{F} \wedge \mathbf{C})^2} - (t-1)^2 (\mathbf{F} \wedge \mathbf{C})^2$$

- If  $d^2 > 0$  when  $t = 0$ , possible larger culling distance



# Fog bank occlusion

- Solution exists at  $t = 0$  when  $k > 1$
- Solution exists deeper when  $k < 27/16$
- Take the larger distance if both exist

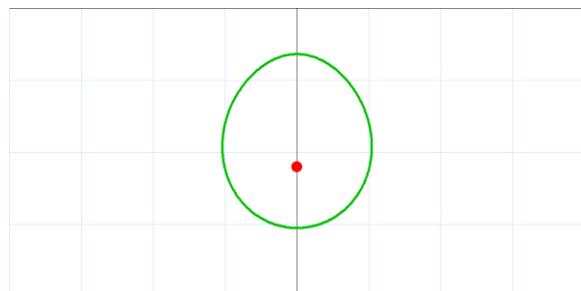
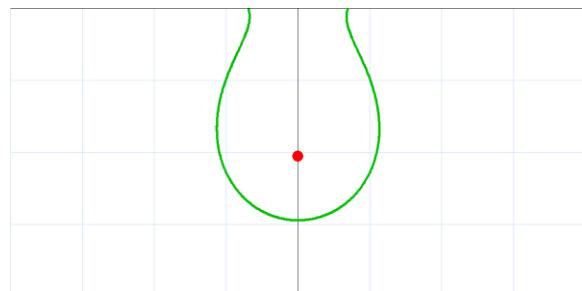
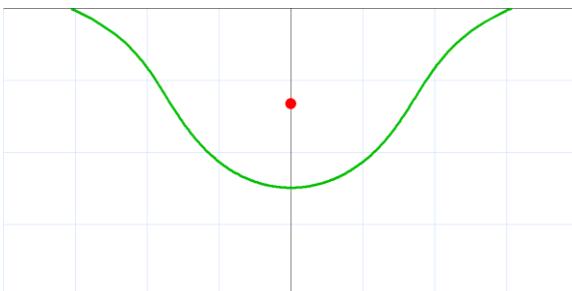


# Fog bank occlusion

$$k > \frac{27}{16}$$

$$k \in \left[1, \frac{27}{16}\right]$$

$$k < 1$$





# Fog bank occlusion

- Strategy:

Eliminate variables

Know what functions look like

Embrace Newton's method



# Contact

- lengyel@terathon.com
- <http://www.terathon.com/lengyel/>
- @EricLengyel



# Supplemental slides

- Bézier animation curves
- Floor and ceiling functions
- Cross product trick
- Bit manipulation tricks



# Bézier animation curves

- Another note about Newton's method
- Cubic 2D Bézier curves often used to animate some component of an object's transform
- Position  $x, y, z$  or rotation angle, for example



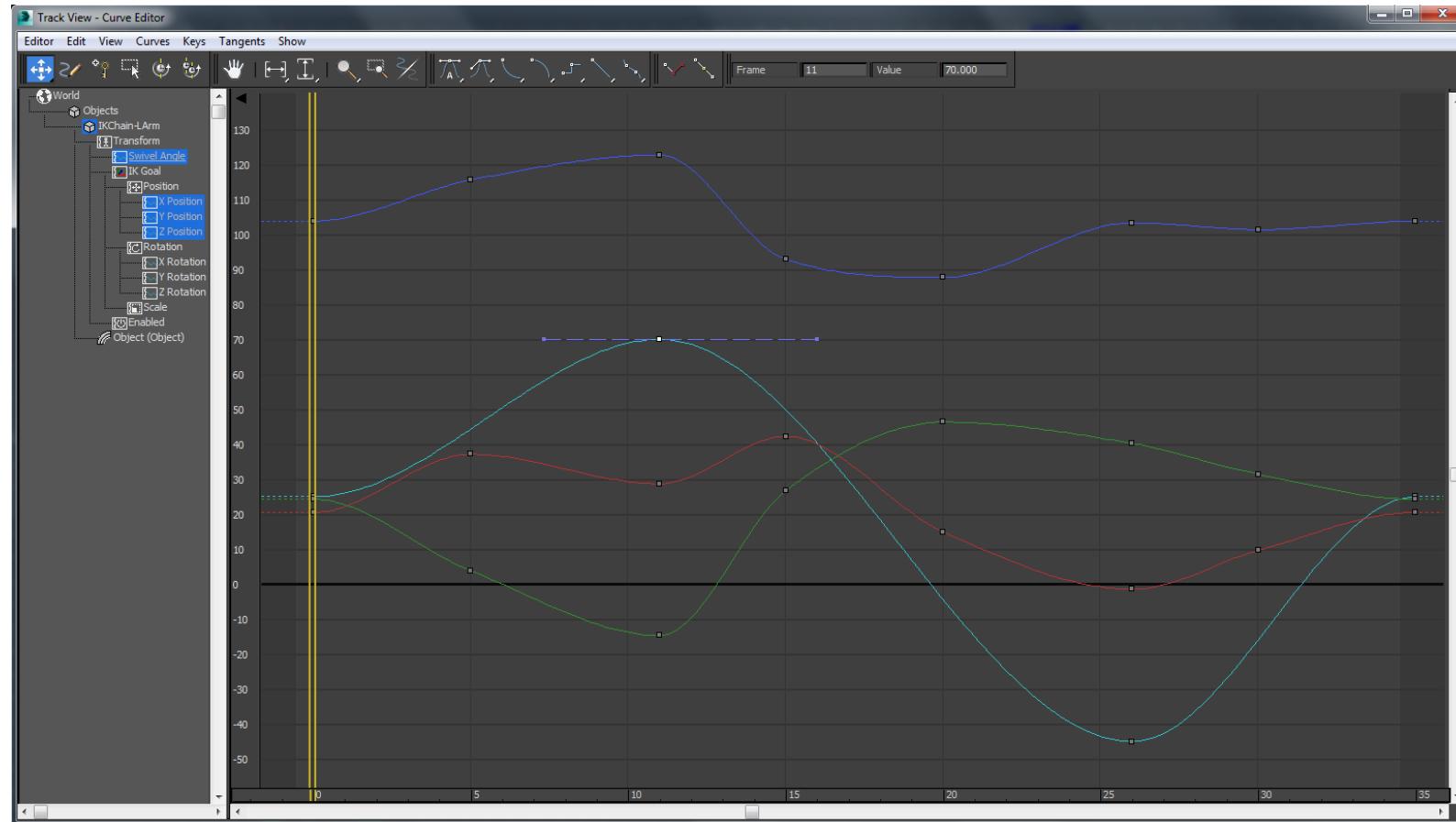
# Bézier animation curves

- 2D coordinates on curve are time  $t$  and some scalar value  $v$
- $t$  is *not* the parameter along the curve
- Big source of confusion in data exchange



# Bézier animation curves

- Control points specified in  $(t, v)$  space
- Time coordinates increase monotonically





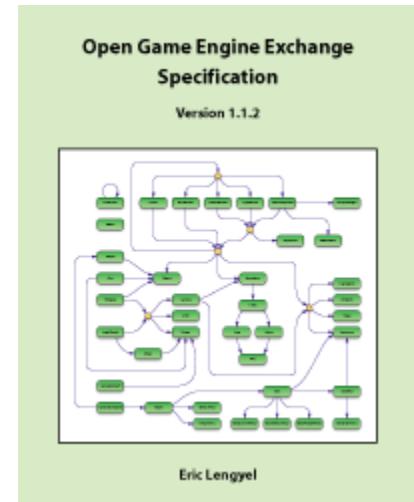
# Bézier animation curves

- To evaluate the value of a curve  $\mathbf{P}(s)$  at a given time  $t$ , it's necessary to find the parameter  $s$  along the curve for which  $P_t(s) = t$
- Requires solving a cubic polynomial
- Newton's method perfect for this case



# Bézier animation curves

- For details, see Track structure in OpenGEX Specification
- [opengex.org](http://opengex.org)





# Floor and ceiling

- Not all CPUs have floating-point floor/ceil/trunc/round instructions
- Need to implement with ordinary math
- Needs to be fast, no FP/int conversions



# Floor function

- 32-bit float has 23 bits in mantissa
- Thus, any value greater than or equal to  $2^{23}$  is necessarily an integer
  - No bits left for any fractional part



# Floor function

- Trick is to add and subtract  $2^{23}$
- The addition causes all fraction bits to be shifted out the right end
- The subtraction shifts zeros back into the space previously occupied by the fraction



# Floor function

- When we add  $2^{23}$ , the original number is rounded to the nearest integer +  $2^{23}$
- If result is greater than original number, then simply subtract one to get floor



# Floor function

- What about negative numbers?
- Use the same trick, but subtract  $2^{23}$  first, and then add it back
- Can combine for all possible inputs



# Floor function

```
__m128 floor(__m128 x)
{
    __m128 one = {0x3F800000};
    __m128 two23 = {0x4B000000};
    __m128 f = _mm_sub_ps(_mm_add_ps(f, two23), two23);
    f = _mm_add_ps(_mm_sub_ps(x, two23), two23);
    f = _mm_sub_ps(f, _mm_and_ps(one, _mm_cmplt_ps(x, f)));
    return (f);
}
```



# Floor function

- But wait, this fails for some very large inputs (bigger than  $2^{23}$ )
- All of these inputs are already integers!
  - They must be if they're bigger than  $2^{23}$



# Floor function

- So just return the input if it's  $> 2^{23}$

```
__m128 sgn = {0x80000000};  
__m128 msk = _mm_cmplt_ps(two23, _mm_andnot_ps(sgn, x));  
f = _mm_or_ps(_mm_andnot_ps(msk, f), _mm_and_ps(msk, x));
```



# Ceiling function

- Instead of subtracting one if result is greater than input, add one if result is less than input

```
f = _mm_add_ps(f, _mm_and_ps(one, _mm_cmplt_ps(f, x)));
```



# Floor and ceiling

- Strategy:

Reduce problem domain



# Cross product trick

- Cross product  $V \times W$  given by:

$$V.yzx * W.zxy - W.zxy * V.yzx$$

- Two mults, one sub, **four** shuffles



# Cross product trick

- Can do this instead:

$$(V * W.yzx - V.yzx * W).yzx$$

- Two mults, one sub, **three** shuffles
  - And same shuffle each time



# Bit manipulation tricks

- Range checks
- Non-branching calculations
- Logic formulas



# Integer range checks

- Integer range checks can always be done with a single comparison:

$$(\text{unsigned}) (x - \text{min}) \leq (\text{unsigned}) (\text{max} - \text{min})$$



# Non-branching calculations

- Using logic tricks to avoid branches in integer calculations
- Many involve using sign bit in clever way
- Also useful to know  $-x == \sim x + 1$



# Non-branching calculations

- Helps scheduling, increases ILP
- Reduces pollution in branch history table
- But can obfuscate code
  - Use where performance is very important
  - Don't bother elsewhere



# Clever uses of sign bit

- `if (a < 0) ++x;`
- Replace with:
- `x -= a >> 31; // 32-bit ints`



# Right-shifting negative integers

- Shifting  $n$ -bit int right by  $n - 1$  bits:
  - All zeros for positive ints (or zero)
  - All ones for negative ints
- C++ standard says `a >> 31` is “implementation-defined” if `a` is negative



# Right-shifting negative integers

- Any sensible compiler generates instruction that replicates sign bit
- To avoid issue in this case, could also use:
- `x += (uint32) a >> 31`



# Predicates for 32-bit signed ints

- $(x == 0)$        $\text{lzcnt}(x) >> 5$
- $(x != 0)$        $(\text{lzcnt}(x) >> 5) ^ 1$
- $(x < 0)$        $(\text{uint32}) x >> 31$
- $(x > 0)$        $(\text{uint32}) -x >> 31$
- $(x == y)$        $\text{lzcnt}(x - y) >> 5$
- $(x != y)$        $(\text{uint32}) ((x - y) | (y - x)) >> 31$
  
- **lzcnt() is leading zero count**



# Absolute value

- $y = x \gg 31$
- $\text{abs}(x) = (x \wedge y) - y$
- Because  $-x = \sim x + 1$   
 $= x \wedge \text{xFFFFFFFF} - \text{xFFFFFFFF}$



# Conditional negation

- Same trick can be used to negate for any bool condition:
- `if (condition) x = -x;`
- `x = (x ^ -condition) + condition`



# Logic Formulas

Formula	Operation / Effect	Notes
$x \& (x - 1)$	Clear lowest 1 bit.	If result is 0, then $x$ is $2^n$ .
$x   (x + 1)$	Set lowest 0 bit.	
$x   (x - 1)$	Set all bits to right of lowest 1 bit.	
$x \& (x + 1)$	Clear all bits to right of lowest 0 bit.	If result is 0, then $x$ is $2^n - 1$ .
$x \& \sim x$	Extract lowest 1 bit.	
$\sim x \& (x + 1)$	Extract lowest 0 bit (as a 1 bit).	
$\sim x   (x - 1)$	Create mask for bits other than lowest 1 bit.	
$x   \sim(x + 1)$	Create mask for bits other than lowest 0 bit.	
$x   \sim x$	Create mask for bits left of lowest 1 bit, inclusive.	
$x ^ \sim x$	Create mask for bits left of lowest 1 bit, exclusive.	
$\sim x   (x + 1)$	Create mask for bits left of lowest 0 bit, inclusive.	
$\sim x ^ (x + 1)$	Create mask for bits left of lowest 0 bit, exclusive.	Also $x \equiv (x + 1)$ .
$x ^ (x - 1)$	Create mask for bits right of lowest 1 bit, inclusive.	0 becomes $-1$ .
$\sim x \& (x - 1)$	Create mask for bits right of lowest 1 bit, exclusive.	0 becomes $-1$ .
$x ^ (x + 1)$	Create mask for bits right of lowest 0 bit, inclusive.	remains $-1$ .
$x \& (\sim x - 1)$	Create mask for bits right of lowest 0 bit, exclusive.	remains $-1$ .

This table from "Bit Hacks for Games", [Game Engine Gems 2](#), A K Peters, 2011.



# Contact

- lengyel@terathon.com
- <http://www.terathon.com/lengyel/>
- @EricLengyel