# 2D Polyhedral Bounds of a Clipped, Perspective-Projected 3D Sphere

Michael Mara Morgan McGuire
NVIDIA and Williams College

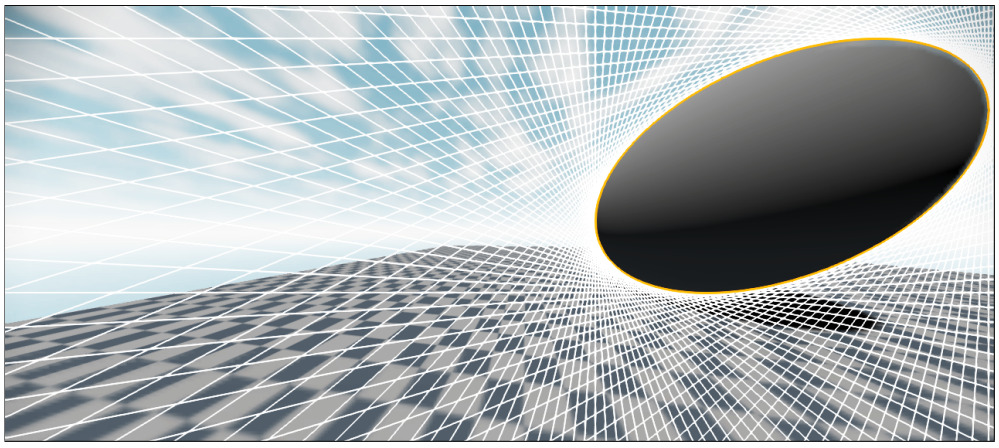

**Figure 1.** A 128-sided 2D bound (orange) of the perspective projection of a sphere, with construction lines (white). We deliberately chose a wide field of view to illustrate that the projection is an ellipse and not a disk.

## Abstract

We show how to efficiently compute 2D polyhedral bounds of the (elliptic) perspective projection of a 3D sphere that has been clipped to the near plane. For the special case of a 2D axis-aligned bounding box, the algorithm is especially efficient.

This has applications for bounding the screen-space effect of an emitter under deferred shading, bounding the kernel in density estimation during image space photon mapping, and bounding the pixel extent of objects for ray casting.

Our solution is designed to elegantly handle the case where the sphere crosses the near clipping plane and efficiently handle all cases on vector processors. In addition to the algorithm, we provide implementations of two common applications: light-tile classification in C++ and expanding an attribute array of spheres (encoded as points and radii) into polygons that cover their silhouettes as a GLSL geometry shader.

## 1. Introduction

### 1.1. Motivating Applications

We encountered the problem of needing a polygonal approximation of the projection of a sphere clipped by the near plane in two related applications. The first was tiled deferred shading. In deferred shading (without tiles), the renderer begins with a screen-space geometry buffer containing, at each pixel, the position, normal, and BSDF coefficients of a point to be shaded. It then iterates over the light sources in the scene and accumulates illumination due to each source at each pixel. One could simply consider the contribution at each pixel due to each light. For many scenes, that contribution is negligible for most lights at every pixel due to radial falloff, so it is far more efficient to iterate over only the pixels that correspond to points within an effective 3D radius of a source. The hardware rasterizer is a power-efficient mechanism for such iteration, so one approach is to rasterize a bounding sphere around the light and then shade geometry-buffer pixels that are covered by that rasterization pass. *Tiled* deferred shading reduces the bandwidth required by processing multiple lights in parallel during each shading pass. It contains a pre-pass that classifies lights into the (e.g., $64 \times 64$ pixel) screen-space tiles by conservative rasterization of the bounding spheres, and then a shading pass that simply applies all lights that affect a tile to all pixels within that tile. We found that some common implementations (that are described in the following section) of the "conservative rasterization" currently employed in the games industry are inefficient, overly conservative, or handled the near clipping plane poorly. The method in this paper efficiently produces a tighter and more robust bound, yet still has a simple implementation. It also generalizes from axis-aligned bounding boxes to arbitrary polygonal bounds.

The kernel-density estimation step of image-space photon mapping is computationally analogous to (untiled) deferred shading with millions of small bounding spheres. The published approach of rasterizing small circumscribed icosahedra both produces overly-conservative projective bounds and requires many polygons to cover ellipses containing potentially few pixels. Since the sphere approximation itself is not shaded, but merely employed as part of a computational solid-geometry approach for covering pixels, the 2D polygonal bound from this paper can yield the same coverage while providing good control over tessellation count and tightness.

We note other motivating applications (for which we do not have personal experience), including computing tight bounds on visible (or proxy) spheres for ray tracing [Williams 2010] and computing bounds for splatting screen-space decals [Kim 2012].

### 1.2. Alternatives

A disk billboard is a reasonable *approximation* of a sphere near the center of the field of view. This approximation fails away from the center, as demonstrated by the top row of Figure 2. The failure occurs because the 2D projection of a 3D sphere's silhouette is not simply a circle with radius proportional to the sphere radius. When the sphere is away from the center of the screen, the silhouette is distorted by perspective into an ellipse. Likewise, note that less than half of the sphere is visible from any scene point due to perspective, so the silhouette of the sphere is never a great circle.

Five common methods for processing the conservative bounds of a projected sphere and related problems are:

*Conservative Box.* Bound the sphere with a cube (i.e., an AABB or OBB) and then clip the cube against the near plane. This creates a convex polyhedron, but only its vertices are used, so the computation is simple. Finally, project all vertices of this polyhedron onto the image plane and compute an axis-aligned bounding box in screen space. We suspect that this is the first ad-hoc approach many would try because it is simple to implement and obviously conservative. Figure 2 shows results from this method using two different cube orientations. In the worst case, the conservative bound contains about twice the area of a tight bound.

*Exhaustive Sphere-Frustum.* Tiled deferred shading needs to conservatively find all screen-space tiles that may overlap the projection of a ball surrounding a light source. One common approach is to geometrically test in 3D whether the ball intersects the frustum of each tile. This is done by culling the sphere against the four bounding planes of each tile. The computation can run in logarithmic time in the number of tiles along a row or column by binary searching for the first plane that culls the sphere along that axis, rather than testing exhaustively [Harada et al. 2012].

*Equation of the Ellipse.* Eberly [1999] solves for the 2D equation of the resulting ellipse in an elegant derivation, and does so for arbitrary ellipsoids, not just spheres. However, this leaves no way of adding the near-plane constraint.

*Axis-Aligned Bounding Box.* Sigg et al. solve for the axis-aligned bounds of a projected ellipsoid, but do not handle the near-plane case (note that all of the images in their paper contain scenes entirely within the view frustum.) [Sigg et al. 2006]

*Intersection of Tangent Planes and Near Plane* Lengyel [2002] computes the formula for planes tangent to the sphere (and parallel to either the $x$- or $y$-axis) and then explicitly intersects them with the view plane to get exact bounds for a scissor region. This derivation is conceptually closest to our own. Lengyel's method requires several fewer operations than ours in the common case. However, on our target architec-

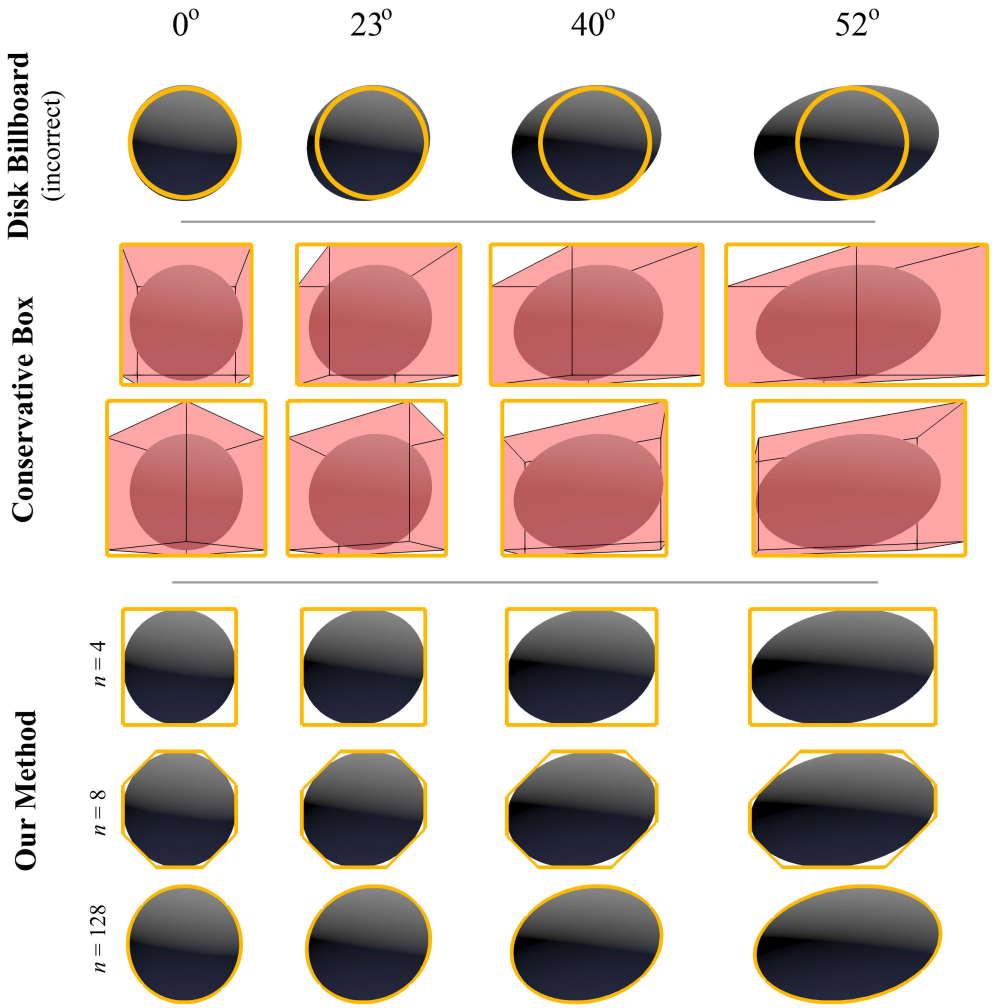


**Figure 2.** 2D silhouette bounds (in thick yellow lines) produced by different algorithms for the black 3D spheres. Each row of the figure is rendered from the same viewpoint, with the center of each sphere 12 degrees above the horizon. The columns show spheres with increasing horizontal angle from the projection axis. The top row shows that the billboard approach is a poor approximation of the sphere's true projection as that angle increases. The simple conservative box approach in the middle rows is correct but produces a loose bound. The bottom three rows show that our algorithm gives arbitrarily tight bounds as $n$, the number of sides, increases.

tures (Kepler-class NVIDIA GPUs), the mixture of operations (e.g., avoid divisions) is more important, so we instead followed a different derivation presented here.

Eberly and Sigg et al. follow a matrix-based symbolic derivation. We give a geometric derivation, like Lengyel, to lend intuition into the nature of the solution. Unlike Lengyel, we directly compute the tangent points and then project onto the view plane. Our approach naturally leads to the desirable support for near-plane clipping.

## 2. Approach

Our geometric approach leads to a fairly elegant solution by addressing each subproblem in the reference frame where it is easiest to solve. Without loss of generality, we initially pose the bounding problem in camera space, where the near clipping plane is $z = z_{\text{near}}$, the view axis is $-\hat{z}$, the screen axes are $\hat{x}$ and $\hat{y}$, and the origin is $\mathcal{O} = (0,0,0)$ (see Figure 3).

We solve the arbitrary bounding polygon problem by first solving for the bounds along an arbitrary axis $\hat{a} = (\cos\phi, \sin\phi, 0)$ in the view plane, and then applying this to multiple axes to carve a 2D convex region that contains the projection of the sphere. The case of solving for an axis-aligned bounding box simplifies dramatically, because it needs to only consider $\hat{a} = (1,0,0)$ and $\hat{a} = (0,1,0)$, thus eliminating cross-terms and trigonometric operations.

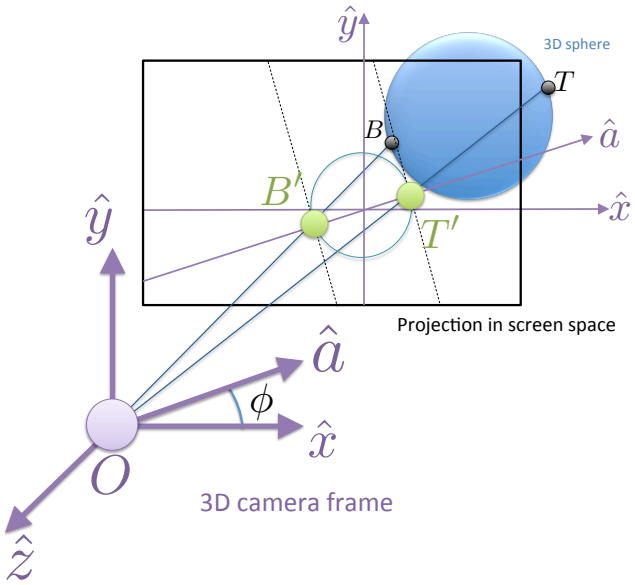


**Figure 3.** Screen-space points $B'$ and $T'$ bound the image of the sphere along the $a$-axis, which is raised from the horizontal by $\phi$. We repeat this for several values of $\phi$ to construct a bounding polygon of arbitrary degree. The math simplifies for the axis-aligned bounding box, where $\hat{a} \in \{\hat{x}, \hat{y}\}$.

In solving for the bounds of the sphere along a single axis $\hat{a}$, we project into the 2D $az$-plane and then find the bounding points $B$ and $T$ that lie on the 1D silhouette of the sphere (which is now a 2D circle). We distinguish 2D points in $az$ and 3D points in $xyz$ with different fonts and use lower-case for vectors and scalars:

- $O = (0,0)$ is the orthographic projection of $\mathcal{O}$ into $az$-space;
- $\mathcal{C} = (x_\mathcal{C}, y_\mathcal{C}, z_\mathcal{C})$ = center of the sphere in camera space;
- $C = (a_C, z_C) = (\hat{a} \cdot (\mathcal{C} - \mathcal{O}), z_\mathcal{C})$ = orthographic projection of $\mathcal{C}$ into $az$-space;
- $\vec{c} = C - O$, vector from the origin $O$ to $C$ in $az$-space;
- $c = ||\vec{c}||$;
- $r$ = radius of the sphere (in both $az$ and $xyz$).

We denote a point produced by perspective projection in any space with a prime (e.g., $T'$).

The silhouette of the 3D sphere is the curve in 3D whose perspective projection into screen space is the 2D ellipse that exactly bounds the projection of the sphere. The silhouette is a circle in 3D (note that the silhouette circle has radius less than $r$, since less than half of the sphere is ever visible under perspective projection.) The silhouette circle also defines a cone in 3D that is exactly tangent to the sphere, with apex $O$ and half-angle of measure $\theta$. This tangency gives rise to our derivation. In the $az$-plane this cone is a triangle.

## 3. Derivation

Figure 4 shows the 2D orthographic projection of the geometry for the single-axis bounding problem into the $az$-plane. The points, $B$ and $T$, are the unknown silhouette

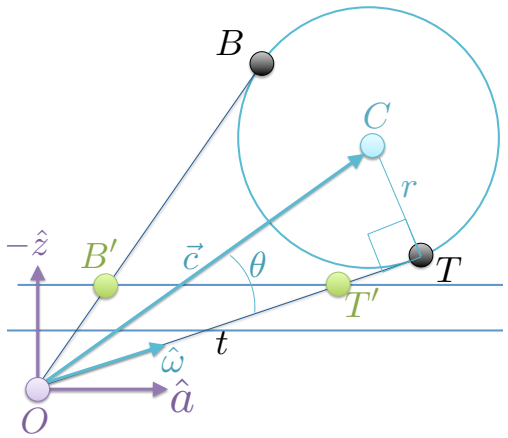


**Figure 4**. Geometry for the derivation of $T$ and $T'$ in the $az$-plane.

points on the sphere, and $B'$ and $T'$ are their projection onto the image plane, which are the points needed for the screen-space bound. For the moment, assume that neither silhouette point is clipped by the near plane.

Let unit vector $\hat{\omega}$ be the direction from $O$ to $T$ and $t$ be the distance between them. We require only an expression for $\hat{w}$ and $t$ to obtain $T$ (the math is symmetric for $B$) and can then transform all points to $xyz$-space to solve the bounding problem without clipping.

### 3.1. Solve for $T$ and $B$

Observe that vector $\hat{\omega}$ is unit vector $\vec{c}/c$ rotated through the cone half-angle whose measure is $\theta$ (for $B$, substitute $-\theta$). Because line $OT$ is tangent to the sphere, $COT$ is a right triangle. Thus,

$$\begin{aligned} t &= \sqrt{c^2 - r^2}, \\ \cos\theta &= \frac{t}{c}, \\ \sin\theta &= \frac{r}{c}, \end{aligned}$$

and we obtain $T$ in $az$-space:

$$\begin{aligned} \hat{\omega} &= \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \vec{c}/c, \\ T &= O + \hat{\omega} t. \end{aligned}$$

The corresponding point $B$ is obtained by using $-\theta$ instead of $\theta$. In practice, since the rotation matrix uses only $\sin\theta$ and $\cos\theta$, we need only negate $\sin\theta$ for our calculation of $B$.

Observe that if $c^2 < r^2$, then the orthogonally projected sphere contains the origin and has no silhouette, so $T$ does not exist.

### 3.2. Clipping to the Near Plane

We now modify our approach to take into account clipping by the near plane. We first consider two simple cases: If $z_C - r > z_{\text{near}}$, the entire sphere is clipped by the near plane and it has no bounds on any axis. If $z_C + r < z_{\text{near}}$, then the bounds are strictly given by $B$ and $T$, so we can skip the clipped bounds computation.

Otherwise, we must consider the intersection of the circle with the near plane (which is just a line in 2D). These occur at

$$\tau, \beta = \left( a_C \pm \sqrt{r^2 - (z_{\text{near}} - z_C)^2}, z_{\text{near}} \right)$$

The smaller solution for the scalar $a$-coordinate gives $\beta$, the larger one gives $\tau$ (see Figure 5.

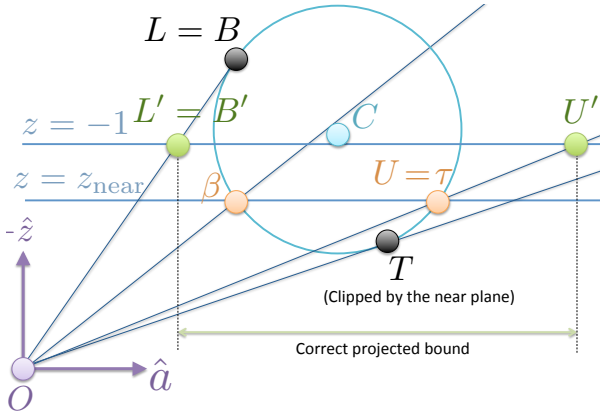


**Figure 5.** The upper-bound point $U$ is $\tau$ when $T$ is clipped by the near plane, and lower bound $L$ is $\beta$ when $B$ is clipped. In this example, $L = B$ and $U = \tau$, because only $\tau$ is clipped by the near plane.

We then test each tangent point against the near plane. The upper bound on the $a$-axis is

$$U = \begin{cases} T & z_T < z_{\text{near}} \\ \tau & \text{otherwise} \end{cases}$$

Let $L$ be the corresponding lower bound, computed using $B$ and $\beta$.

Take each bound back to camera space:

$$\mathcal{U} = (a_U x_a,\ a_U y_a,\ z_U),$$

and then project onto the desired $z$-plane using a single division. If you wish to obtain screen coordinates instead you can simplify the code at the expense of some run-time cost by just applying a point-to-pixel matrix.

## 4. The Bounding Polygon

We note that for a given $\phi$, the $L$ calculated along the corresponding axis $\hat{a} = (\cos\phi, \sin\phi, 0)$ is identical to the $U$ calculated along the axis corresponding to $\phi + \pi$. Thus, when we calculate the bounding points, we can uniquely identify them with the angle $\phi$ for which they are $U$. In order to get an $n$-gon bound, we calculate the bounds for $n/2$ equally-spaced values of $\phi$ on the interval $[0, \pi)$ and then project those bounds into screen space.

For each bounding point $U_\phi$ in the screen plane, we construct a line $\ell_\phi$ through the point and perpendicular to the axis $a$ used to compute the point. We then iterate through these points in order of increasing angle, computing the intersection point $p_\phi$ of $\ell_\phi$ and $\ell'_\phi$ for each $\phi$, where $\phi'$ is the next angle to be visited. The vertices of the bounding $n$-gon are the points $p_\phi$ in iteration order. The process simplifies for for the case of a bounding box, as shown in Listing 2.

## 5. Examples

We give two sample applications of our conservative sphere bounds to common graphics problems. We formatted the source code in the paper to align with the derivation given here and fit on the document pages. The downloadable version follows more verbose and maintainable coding conventions. Both versions use the G3D Innovation Engine version 9.00 beta 4 (http://g3d.sf.net) for support code; we assume the reader will substitute his or her own low-level classes for points, vectors, and so on.

Listing 1 (`lst:getBoundsForAxis`) implements the core of our algorithm, computing the two clipped tangent points along an arbitrary axis. That listing is in C++ for a CPU, but the code is almost identical in GLSL–only the syntax of type declarations changes, e.g., "`const Vector3& a`" to "`in vec3 a`", and in the GLSL code, we pass in $\phi$ and then calculate the axis $a$ instead of passing in $a$ directly. This code is used by both sample applications.

The first usage example is CPU code in C++ implementing the bucketing pre-pass of tiled shading. It iterates over the effective sphere bounding each light source and computes a 2D axis-aligned bounding box, then adds the light to each tile that it could affect. The code in Section 5.2 is in two parts. Listing 2 computes an axis-aligned bounding box in pixel coordinates from four points generated by calling our main algorithm on the $x$- and $y$-axes. Listing 3 (`tileClassification`) demonstrates the outer-tile classification loop using these routines.

The second example is a GPU geometry shader in GLSL that computes a 2D polygonal silhouette approximation. It calculates the polygonal approximation for each of a series of spheres passed as vertex attributes. It solves for the lines bounding the projection and then computes the vertices by intersecting adjacent bounding lines. Finally, it emits a triangle strip that covers the projection of each the sphere. Listing 4 is the support code and Listing 5 is the actual shader body.

### 5.1. Bounds on Arbitrary Axis

```
void getBoundsForAxis
 (const Vector3& a,      // Bounding axis (camera space)
  const Point3&  C,      // Sphere center (camera space)
  float          r,      // Sphere radius
  float          nearZ,  // Near clipping plane (negative)
  Point3&        L,      // Tangent point (camera space)
  Point3&        U) {    // Tangent point (camera space)

  const Vector2 c(dot(a, C), C.z); // C in the a-z frame

  Vector2 bounds[2]; // In the a-z reference frame

  const float tSquared = dot(c, c) - square(r);
  const bool cameraInsideSphere = (tSquared <= 0);

  // (cos, sin) of angle theta between c and a tangent vector
  const Vector2& v = cameraInsideSphere ?
      Vector2(0.0f, 0.0f) :
      Vector2(sqrt(tSquared), r) / c.length();

  // Does the near plane intersect the sphere?
  const bool clipSphere = (c.y + r >= nearZ);

  // Square root of the discriminant; NaN (and unused)
  // if the camera is in the sphere
  float k = sqrt(square(r) - square(nearZ - c.y));

  for (int i = 0; i < 2; ++i) {
    if (! cameraInsideSphere)
      bounds[i] = Matrix2(v.x, -v.y,
                          v.y,  v.x) * c * v.x;

    const bool clipBound =
      cameraInsideSphere || (bounds[i].y > nearZ);

    if (clipSphere && clipBound)
      bounds[i] = Vector2(projCenter.x + k, nearZ);

    // Set up for the lower bound
    v.y = -v.y;  k = -k;
  }

  // Transform back to camera space
  L = bounds[1].x * a;  L.z = bounds[1].y;
  U = bounds[0].x * a;  U.z = bounds[0].y;
}
```

**Listing 1**. Projected 2D bounds along an arbitrary axis.

### 5.2. C++ Light-Tile Classification

```
Vector3 project(const Matrix4& P, const Point3& Q) {
    const Vector4& v = P * Vector4(Q, 1.0f))
    return v.xyz() / v.w;
}

AABox2D getAxisAlignedBoundingBox
   (const Point3& C,      // camera-space sphere center
    float r,              // sphere radius
    float nearZ,          // near clipping plane position (negative)
    const Matrix4& P) {   // camera to screen space

    // Points on edges
    Point3 right, left, top, bottom;
    getBoundsForAxis(Vector3(1,0,0), C, r, nearZ, left,   right);
    getBoundsForAxis(Vector3(0,1,0), C, r, nearZ, bottom, top);

    return AABox2D::xyxy(project(P, left).x,  project(P, bottom).y,
                         project(P, right).x, project(P, top).y);
}
```

**Listing 2**. Axis-aligned bounding box.

```
/** Center is in camera space */
void tileClassification(int tileNumX, int tileNumY,
                        int tileWidth, int tileHeight,
                        const Point3& C, float r,
                        float nearZ,
                        const Matrix4& projMatrix){
    Rect2D projectedBounds = getBoundingBox(C, r,
                             projectionMatrix);

    int minTileX = max(0,               (int)(projectedBox.x0() / tileWidth));
    int maxTileX = min(tileNumX - 1,    (int)(projectedBox.x1() / tileWidth));

    int minTileY = max(0,               (int)(projectedBox.y0() / tileHeight));
    int maxTileY = min(tileNumY - 1,    (int)(projectedBox.y1() / tileHeight));

    for(int i = minTileX; i <= maxTileX; ++i){
        for(int j = minTileY; j <= maxTileY; ++j){
            // This tile is touched by the bounding box
            // of the sphere
            // Put application specific tile-classification
            // code here.
        }
    }
}
```

**Listing 3**. Tile classification.

### 5.3. GLSL Ellipse Geometry Shader

```
#version 330 // "Optimized" for NVIDIA Kepler (6xx and 7xx) GPUs for N <= 12
#define N (8) // Even number of sides of bounding polygon
#define NUM_AXES (N/2)

layout(points) in;
layout(triangle_strip, max_vertices = N) out;

in float radius[]; // The radius of the sphere.  Only one element in the array

uniform mat4 projMatrix;
uniform float nearZ;

struct line2D {
    vec2 point;
    vec2 direction;
};

float square(float x) { return x*x; }

// Assume not parallel
vec2 intersect(line2D L1, line2D L2){
    float denominator = (L1.direction.x * L2.direction.y) -
                        (L1.direction.y * L2.direction.x);
    float leftTerm  =   (L1.point.x + L1.direction.x) * L1.point.y -
                        (L1.point.y + L1.direction.y) * L1.point.x;
    float rightTerm =   (L2.point.x + L2.direction.x) * L2.point.y -
                        (L2.point.y + L2.direction.y) * L2.point.x;

    vec2 numerator = leftTerm * L2.direction - rightTerm * L1.direction;

    return numerator / denominator;
}
```

**Listing 4**. Support routines for the shader.

```
void main() {
    float r  = radius[0];
    vec3 C = gl_in[0].gl_Position.xyz;

    if (C.z - r >= nearZ) return; // culled by near plane

    vec3    axesBounds[N];
    line2D  boundingLines[N + 1];
    float invAxisNum = 1.0 / NUM_AXES;

    for (int i = 0; i < NUM_AXES; ++i) {
        float phi = (i * PI) * invAxisNum;
        getBoundsForPhi(phi, C, r, nearZ, boundingLines[i].direction,
                    axesBounds[i], axesBounds[i + NUM_AXES]);
        boundingLines[i + NUM_AXES].direction = boundingLines[i].direction;
    }

    // The z-coordinate for the intersections
    float maxZ = min(nearZ, C.z + r);

    for (int i = 0; i < N; ++i)
        boundingLines[i].point = axesBounds[i].xy * (maxZ / axesBounds[i].z);

    // Duplicate the first bounding line as the last to avoid a modulo operation
    boundingLines[N] = boundingLines[0];

    // Emit a triangle strip
    for (int i = 0; i < AXIS_NUM; ++i) {
        int j = N - i - 1;
        vec4 pos = vec4(intersect(boundingLines[j], boundingLines[j+1]), maxZ, 1.0f);
        gl_Position = projMatrix * pos;
        EmitVertex();
        pos = vec4(intersect(boundingLines[i], boundingLines[i+1]), maxZ, 1.0f);
        gl_Position = projMatrix * pos;
        EmitVertex();
    }
    EndPrimitive();
}
```

**Listing 5.** Geometry shader that expands parametric spheres into triangle strips bounding their silhouettes.

## References

EBERLY, D. 1999. Perspective projection of an ellipsoid. Geometric Tools, LLC. http://www.geometrictools.com/Documentation/PerspectiveProjectionEllipsoid.pdf. 72

HARADA, T., MCKEE, J., AND YANG, J. C. 2012. Forward+: Bringing deferred lighting to the next level. In *Eurographics Short Paper*, Eurographics Association, Aire-la-Ville, Switzerland, 5–8. 72

KIM, P. 2012. Screen space decals in Warhammer 40,000: Space Marine. In *ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA, SIGGRAPH '12, 6:1–6:1. 71

LENGYEL, E. 2002. The mechanics of robust stencil shadows. Gamasutra.com. Available at http://www.gamasutra.com/the_mechanics_of_robust_stencil_.php. 72

SIGG, C., WEYRICH, T., BOTSCH, M., AND GROSS, M. H. 2006. GPU-based ray-casting of quadratic surfaces. In *SPBG*, Eurographics Association, Aire-la-Ville, Switzerland, M. Botsch, B. Chen, M. Pauly, and M. Zwicker, Eds., 59–65. 72

WILLIAMS, M., 2010. Big shiny balls. Tech Report on Rendering in the video game Hustle Kings, http://www.voofoostudios.com/?p=33. 71

## Author Contact Information

Michael Mara (mmara@nvidia.com)
Morgan McGuire (morgan@cs.williams.edu)
http://graphics.cs.williams.edu
47 Lab Campus Drive
Williamstown, MA 01267

---

Mara and McGuire, 2D Polyhedral Bounds of a Clipped, Perspective-Projected 3D Sphere, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, 70–83, 2013
http://jcgt.org/published/0002/02/05/

| | | | |
|---|---|---|---|
| Received: | 2013-01-02 | | |
| Recommended: | 2013-02-20 | Corresponding Editor : | Eric Haines |
| Published: | 2013-08-22 | Acting Editor-in-Chief: | Eric Haines |