View Frustum Optimization
To Maximize Object's Image Area

Kok-Lim Low Adrian Ilie

Department of Computer Science
University of North Carolina at Chapel Hill
Email: {lowk, adyilie}@cs.unc.edu

ABSTRACT

This paper presents a method to compute a view frustum for a 3D object viewed from a given viewpoint, such that the object is completely enclosed in the frustum and the final object's image area is also *near-maximal* in the given 2D rectangular viewing region. This optimization can be used to improve the resolution of shadow maps and texture maps for projective texture mapping. Instead of doing the optimization in 3D space to find a good view frustum, our method uses a 2D approach. The basic idea of our approach is as follows. First, from the given viewpoint, a sample image of the object is generated using a conveniently-computed view frustum. A tight 2D bounding quadrilateral is then computed to enclose the image of the object. Next, considering the projective warp between the bounding quadrilateral and the rectangular viewing region, our method applies a technique of camera calibration to compute a new view frustum that generates an image that covers the viewing region as much as possible.

1 INTRODUCTION

In interactive computer graphics rendering, we often need to compute a *view frustum* from a given viewpoint such that a selected 3D object or a group of 3D objects is totally inside the rendered 2D rectangular image. This kind of view-frustum computation is usually needed when generating *shadow maps* [Williams78] from light sources, and images for *projective texture mapping* [Segal92, Hoff98].

The easiest way to compute such a view frustum is to pre-compute a simple 3D bounding volume, such as a bounding sphere, around the 3D object, and create a *symmetric* perspective view frustum that encloses the object's bounding volume. However, very often, this view frustum is not enclosing the 3D object tightly enough to produce an image of the object that covers the 2D rectangular viewing region as much as possible. We will refer to the image of the object as the *object's image*, and the 2D rectangular viewing region as the *viewport*. If the object's image is too small, we are not efficiently utilizing the available viewport area to produce a shadow map or projective texture map that could have higher-resolution due to a larger image of the object. A small image region of the object in a shadow map usually results in blocky shadow edges, and similarly, a low-resolution image region in a texture map can also result in a blocky rendered image.

Other methods increase the object's image area in the viewport by using a tighter 3D bounding volume, such as the 3D convex hull of the object [Berg97]. However, this is computationally expensive, and there is still a lot of room for improvement by manipulating the shape of the view frustum and the orientation of the image plane. Figure 1 shows an example.

This paper presents a method to compute a view frustum for a 3D object viewed from a given viewpoint, such that the final object's image is entirely inside the viewport and its area is also *near-maximal*. For computational efficiency, our method does not seek to compute the optimal view frustum, but to compromise for one that is near-optimal.

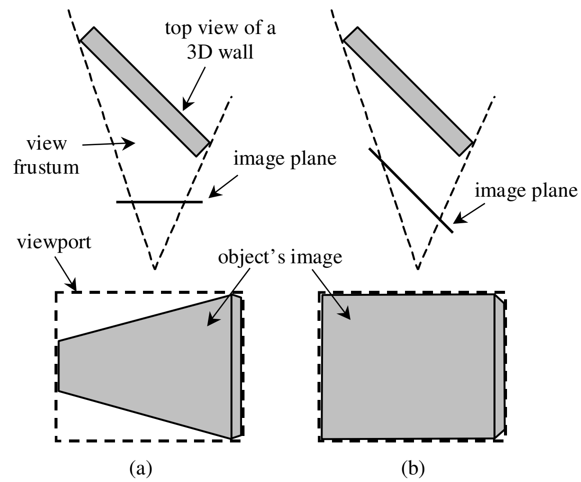

Figure 1*: (a) The symmetric perspective view frustum cannot enclose the 3D object tightly enough, therefore, the object's image does not efficiently utilize the viewport area. (b) By manipulating the view frustum such that the image plane becomes parallel to the larger face of the 3D wall, we can improve the object's image to cover almost the whole viewport.*

Instead of doing the optimization in 3D space to find a good view frustum, our method uses a 2D approach. This makes the method more efficient and simpler to implement. The basic idea of our approach is as follows. First, from the given viewpoint, a sample image of the whole object is generated using a conveniently-computed view frustum. From the sample image, our method uses a tight 2D bounding quadrilateral of the image to decide how the image could be warped to fill the viewport as much as possible. Then by applying a technique of camera calibration from the field of computer vision [Faugeras93, Trucco98], the image warping information is used to derive a valid view frustum, such that it will generate the target image to fill the viewport as much as possible.

Contributions

One of the main contributions of this work is the recognition of the 2D relationship between the images of an object in different image planes, and also the relationship between these image planes and their view frusta. This allows us to efficiently perform the optimization in 2D, and then transform the result into a valid frustum. We also introduce the use of a well-studied camera calibration technique in computer vision to derive the desired view frustum. Another contribution is the introduction of an efficient algorithm to compute a near-optimal tight bounding quadrilateral enclosing a set of 2D points.

Paper Outline

In the next section, we describe how a view frustum is defined in the context of the OpenGL API and provide an overview of our method. Section 3 describes in detail our algorithm to compute a tight bounding quadrilateral enclosing a set of 2D points, and Section 4 details the use of a camera calibration technique to derive a desired view frustum. We show some of our results in Section 5, and discuss some issues related to our method in Section 6. Finally, we conclude the paper in Section 7.

2 OVERVIEW OF METHOD

Without loss of generality, we will describe our method in the context of the OpenGL API [Woo99]. We expect the readers have already had experience with the OpenGL API (or similar APIs), so OpenGL serves as the common unambiguous specification on which our descriptions are based. This allows easier and clearer explanation. Moreover, those readers who use OpenGL can quickly implement the method without additional modification and conversion.

In OpenGL, defining a view frustum from an arbitrary viewpoint requires the definition of two transformations. The first is the *view transformation*, and it transforms points in the world coordinate system into the eye coordinate system. The second transformation is the *projection transformation*, and it transforms points in the eye coordinate system into the normalized device coordinate (NDC) system.

Given a viewpoint, a 3D object in the world coordinate system, and the viewport's width and height, our objective is to compute a valid view frustum (i.e. a view transformation and a projection transformation) that maximizes the area of the object's image in the viewport. We provide an overview of our method below.

Generate Sample Image

We generate a sample image of the entire object, as seen from the viewpoint, using a conveniently-computed view frustum. This view frustum can be easily computed by bounding the object with a sphere and then creating a symmetric perspective view frustum that encloses the sphere. The view transformation and the projection transformation that represent the symmetric view frustum can be readily obtained from the OpenGL API.

We do not actually render a 2D image of the object using this view frustum. Instead, we use the two transformations and the viewport settings to explicitly transform all the 3D vertices of the object from the world coordinate system into the 2D window coordinate system. Effectively, we project the 3D vertices of the object onto their corresponding 2D image points.

Compute Tight Bounding Quadrilateral

We compute a tight bounding quadrilateral of the 2D image points by first computing a 2D convex hull of the 2D image points, and then incrementally decimating the edges of the convex hull until a bounding quadrilateral remains. Figure 2 shows an example.

The most important idea of our method lies in the observation that the bounding quadrilateral and the rectangular viewport are related only by a projective warp or 2D collineation (see Chapter 2 of [Faugeras93]). Equally important to know is that this

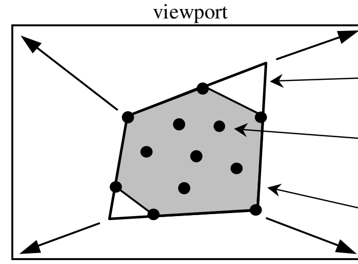

Figure 2*: The basic idea of our method. The 3D vertices of the object are first projected onto their corresponding 2D image points. A 2D convex hull is computed for these image points, and it is then incrementally reduced to a quadrilateral. The bounding quadrilateral is related to the viewport's rectangle by a projective warp. This warping effect can be achieved by rotating and moving the image plane.*

projective warp from the bounding quadrilateral to a rectangle can be achieved by merely rotating and moving the image plane.

Section 3 gives more details about our method of computing a tight bounding quadrilateral.

Compute View Frustum

We want to compute a view frustum whose near and far planes are oriented in such a way with respect to the object that the bounding quadrilateral is warped into the viewport's rectangle.

We first project each corner of the bounding quadrilateral back into the 3D world coordinate system as a ray originating from the viewpoint. Taking the world coordinates of any 3D point on each ray and pairing it with the 2D window coordinates of the corresponding corner of the viewport's rectangle, we get a pair-correspondence. With four pair-correspondences, one for each corner, we are able to use a camera calibration technique to solve for the desired view frustum. The details of the computation are given in Section 4.

3 COMPUTING TIGHT BOUNDING QUADRILATERAL

Aggarwal et al. presented an $O(n^2 \log n \log k)$ algorithm to compute the smallest convex k-sided polygon to enclose a given convex n-sided polygon [Aggarwal85]. For our case of computing a convex bounding quadrilateral, $k = 4$, and the time complexity of their algorithm becomes $O(n^2 \log n)$. To use their algorithm, we would first need to compute a 2D convex hull of the 2D image points in the window coordinate system. However, if the convex hull is complex (n is large), a super-quadratic algorithm is not likely to be efficient enough for interactive graphics applications in which the viewpoint (or light source's position) changes very often. Moreover, the algorithm can be difficult to implement.

Here, we propose an alternative algorithm to compute a convex bounding quadrilateral. The result produced by our algorithm is only near-optimal, however the algorithm has time complexity $O(n \log n)$.

Our algorithm obtains the convex bounding quadrilateral by iteratively eliminating edges from the convex hull using a greedy approach until only four edges remain. Suppose the convex hull of the 2D image points has n sides, and they are numbered from 0 to $n - 1$. To eliminate an edge i, we need to first make sure that the sum of the interior angles it makes with the two adjacent edges is more than 180°. Then, we extend the two adjacent edges towards each other to intersect at a point (see Figure 3).

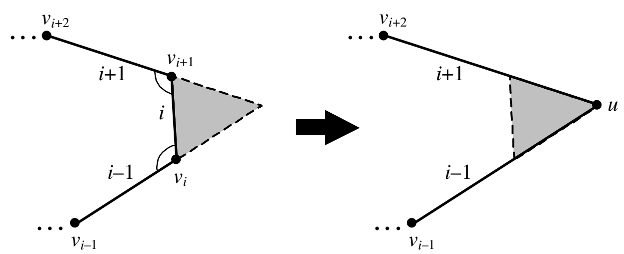

***Figure 3**: Eliminating edge i.*

During each iteration, we choose to eliminate the edge whose removal will add the smallest area to the resulting polygon. For example, in Figure 3, removing edge i will add the gray-shaded area to the resulting polygon. This edge-removal operation is done until the resulting polygon becomes a quadrilateral. It can be easily proved that for any convex polygon of five or more sides, there always exists at least one edge that can be removed (see the proof in Appendix). Since the resulting polygon is also a convex polygon, by induction, we can always reduce the initial input convex hull to a convex quadrilateral.

Of course, if the initial convex hull is already a quadrilateral, we do not need to do anything. If the initial convex hull is a triangle, we just create a bounding parallelogram whose diagonal corresponds to the longest edge of the triangle, and three of its corners coincide with the three corners of the triangle. This ensures that the object's image wi ll occupy half the viewport.

Complexity Analysis

If the number of 2D image points is m, then their convex hull can be computed in $O(m \log m)$ time [Berg97]. Let the number of vertices on the convex hull be n. Our algorithm can compute a bounding quadrilateral in $O(n \log n)$ time. To achieve that, we use a heap to keep track of the area that would be added by the removal of each edge. After an edge is removed, only the added areas for its two neighboring edges need to be updated.

4 COMPUTING VIEW FRUSTUM

After we have found a tight bounding quadrilateral, we want to compute a view frustum that warps the quadrilateral to the viewport's rectangle as illustrated in Figure 2.

First, we need to decide to which corner of the viewport's rectangle each quadrilateral corner is to be warped. We have chosen to match the longest edge and its opposite edge of the quadrilateral with the longer edges of the viewport's rectangle.

Using the view transformation and the projection transformation of the conveniently-computed view frustum, we inverse-project each corner of the bounding quadrilateral back into the 3D world coordinate system as a ray originating from the viewpoint. Taking the world coordinates of any 3D point on the ray and pairing it with the 2D *pixel coordinates* of the corresponding corner of the viewport's rectangle, we get a *pair-correspondence*. With four pair-correspondences, one for each corner, we are able to use a camera calibration technique to solve for the desired view frustum. In other words, we are computing a new view transformation and a new projection transformation.

4.1 A Camera Calibration Technique

For a pinhole camera, which is the camera model used in OpenGL, the effect of transforming a 3D point in the world coordinate system into a 2D image point in the viewport can be described by the following expression:

$$\begin{pmatrix} u_i \\ v_i \\ w_i \end{pmatrix} = \mathbf{P} \cdot \begin{pmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & c_x \\ 0 & b & c_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \cdot \begin{pmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{pmatrix} \quad (1)$$

where

- a, b, c_x and c_y are collectively called the *intrinsic parameters* of the camera,
- r_{ij} and t_i respectively define the *rotation* and *translation* of the view transformation, and they are called the *extrinsic parameters* of the camera,
- $(X_i, Y_i, Z_i, 1)^T$ are the homogeneous coordinates of a point in the world coordinate system, and
- the pixel coordinates of the 2D image point are

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} u_i / w_i \\ v_i / w_i \end{pmatrix}. \quad (2)$$

$\mathbf{P}$ is a 3×4 matrix called a *projection matrix*. This projection matrix is not the same as the OpenGL projection transformation mentioned in Section 2. The former maps a 3D point in the world coordinate system to 2D pixel coordinates, whereas the latter maps a 3D point in the eye coordinate system to a 3D point in the NDC. From here onwards, we will refer to a matrix representing the latter as an *OpenGL projection matrix*.

Since the viewpoint's position is known, we can first apply a translation to the world coordinate system such that the viewpoint is now located at the origin. We will refer to this as the *shifted world coordinate system*, and with it, we can simplify (1) to

$$\begin{pmatrix} u_i \\ v_i \\ w_i \end{pmatrix} = \mathbf{P} \cdot \begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} = \begin{pmatrix} a & 0 & c_x \\ 0 & b & c_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \cdot \begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} \quad (3)$$

where $\mathbf{P}$ is now a 3×3 matrix, and $(X_i, Y_i, Z_i)^T$ are the 3D coordinates of a point in the shifted world coordinate system.

To solve for the intrinsic and extrinsic camera parameters, we will first solve for the matrix $\mathbf{P}$, and then decompose $\mathbf{P}$ into the individual camera parameters.

4.1.1 Solving for the Projection Matrix

If we write $\mathbf{P}$ as

$$\mathbf{P} = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} \quad (4)$$

then the pixel coordinates of the ith 2D image point can be written as

$$x_i = \frac{u_i}{w_i} = \frac{p_{11}X_i + p_{12}Y_i + p_{13}Z_i}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i}$$
$$y_i = \frac{v_i}{w_i} = \frac{p_{21}X_i + p_{22}Y_i + p_{23}Z_i}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i}. \tag{5}$$

We can rearrange (5) to get

$$p_{11}X_i + p_{12}Y_i + p_{13}Z_i - x_i(p_{31}X_i + p_{32}Y_i + p_{33}Z_i) = 0$$
$$p_{21}X_i + p_{22}Y_i + p_{23}Z_i - y_i(p_{31}X_i + p_{32}Y_i + p_{33}Z_i) = 0. \tag{6}$$

Because of the divisions u_i/w_i and v_i/w_i in (5), **P** can be multiplied by any non-zero scalar and (x_i, y_i) will still remain the same. **P** is said to be defined up to an arbitrary scale factor, and has only eight independent entries. Therefore, the four pair-correspondences we have previously obtained are sufficient to solve for **P**. Note that because of the removal of the translation in (3), the 3D point in each pair-correspondence must now be translated into the shifted world coordinate system. To prevent degeneracy, no three corners of the bounding quadrilateral should be collinear.

With the four pair-correspondences, we can form a homogeneous linear system

$$\mathbf{A} \cdot \mathbf{p} = \mathbf{0} \tag{7}$$

where

$$\mathbf{p} = (p_{11}, p_{12}, p_{13}, p_{21}, p_{22}, p_{23}, p_{31}, p_{32}, p_{33})^{\mathrm{T}} \tag{8}$$

and

$$\mathbf{A} = \begin{pmatrix} X_1 & Y_1 & Z_1 & 0 & 0 & 0 & -x_1X_1 & -x_1Y_1 & -x_1Z_1 \\ 0 & 0 & 0 & X_1 & Y_1 & Z_1 & -y_1X_1 & -y_1Y_1 & -y_1Z_1 \\ X_2 & Y_2 & Z_2 & 0 & 0 & 0 & -x_2X_2 & -x_2Y_2 & -x_2Z_2 \\ 0 & 0 & 0 & X_2 & Y_2 & Z_2 & -y_2X_2 & -y_2Y_2 & -y_2Z_2 \\ X_3 & Y_3 & Z_3 & 0 & 0 & 0 & -x_3X_3 & -x_3Y_3 & -x_3Z_3 \\ 0 & 0 & 0 & X_3 & Y_3 & Z_3 & -y_3X_3 & -y_3Y_3 & -y_3Z_3 \\ X_4 & Y_4 & Z_4 & 0 & 0 & 0 & -x_4X_4 & -x_4Y_4 & -x_4Z_4 \\ 0 & 0 & 0 & X_4 & Y_4 & Z_4 & -y_4X_4 & -y_4Y_4 & -y_4Z_4 \end{pmatrix} \tag{9}$$

For the homogeneous system $\mathbf{A}\cdot\mathbf{p} = \mathbf{0}$, the vector **p** can be computed using SVD (singular value decomposition) related techniques as the eigenvector corresponding to the only zero eigenvalue of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. In other words, if the SVD of **A** is $\mathbf{UDV}^{\mathrm{T}}$, then **p** is the column of **V** corresponding to the only zero singular value of **A**. For more details about camera calibration, see [Trucco98], and for a comprehensive introduction to linear algebra and SVD, see [Strang88]. An implementation of SVD can be found in [Press93].

4.1.2 Computing Camera Parameters

From the computed projection matrix, we want to express the intrinsic and extrinsic parameters as closed-form functions of the matrix entries. Since **P** is defined up to an arbitrary scale factor, the computed matrix may differ from the theoretical **P** by a scale factor. We now let **Q** be the computed matrix. The matrix **P** can be expressed in terms of the parameters as

$$\mathbf{P} = \begin{pmatrix} ar_{11} + c_x r_{31} & ar_{12} + c_x r_{32} & ar_{13} + c_x r_{33} \\ br_{21} + c_y r_{31} & br_{22} + c_y r_{32} & br_{23} + c_y r_{33} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}. \tag{10}$$

Then, we can write

$$\mathbf{Q} = \lambda\alpha\mathbf{P} \tag{11}$$

where $\lambda = \pm 1$ is the sign of the scale factor, and $\alpha > 0$ is the absolute value of the scale factor.

We observe that the last row of **P** corresponds to the last row of the rotation matrix. Using the fact that every row of a rotation matrix is a unit vector, we can find the absolute value of the scale factor as

$$\alpha = \alpha\sqrt{r_{31}^2 + r_{32}^2 + r_{33}^2} = \sqrt{q_{31}^2 + q_{32}^2 + q_{33}^2} \tag{12}$$

where each q_{ij} is an entry of **Q**.

We normalize **Q** by dividing each of its entries by α. From here onwards, **Q** refers to the normalized matrix and q_{ij} are its normalized entries. To help in the following derivations, we first define the following 3D vectors:

$$\mathbf{q}_1 = (q_{11}, q_{12}, q_{13})^{\mathrm{T}},$$
$$\mathbf{q}_2 = (q_{21}, q_{22}, q_{23})^{\mathrm{T}}, \tag{13}$$
$$\mathbf{q}_3 = (q_{31}, q_{32}, q_{33})^{\mathrm{T}}.$$

The values of the parameters can be computed as follows:

$$c_x = \mathbf{q}_1^{\mathrm{T}}\mathbf{q}_3,$$
$$c_y = \mathbf{q}_2^{\mathrm{T}}\mathbf{q}_3,$$
$$a = -\sqrt{\mathbf{q}_1^{\mathrm{T}}\mathbf{q}_1 - c_x^2},$$
$$b = -\sqrt{\mathbf{q}_2^{\mathrm{T}}\mathbf{q}_2 - c_y^2}, \tag{14}$$
$$(r_{11}, r_{12}, r_{13})^{\mathrm{T}} = (\mathbf{q}_1 - c_x\mathbf{q}_3)/a,$$
$$(r_{21}, r_{22}, r_{23})^{\mathrm{T}} = (\mathbf{q}_2 - c_y\mathbf{q}_3)/b,$$
$$(r_{31}, r_{32}, r_{33})^{\mathrm{T}} = \mathbf{q}_3.$$

The sign λ affects only the values of r_{ij}. It can be determined as follows. First, we use the rotation matrix $[r_{ij}]$ computed in the above procedure to transform the 4 shifted world points in the pair-correspondences. Since these 3D points are all in front of the camera, their transformed z-coordinates should be negative, because the camera is looking in the $-z$ direction in the eye coordinate system. If it is not the case, we correct the r_{ij} by changing their signs.

4.1.3 Conversion to OpenGL Matrices

From the camera parameters obtained above, the OpenGL view transformation matrix is

$$\mathbf{M}_{\mathrm{MODELVIEW}} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & (r_{11}v_x + r_{12}v_y + r_{13}v_z) \\ r_{21} & r_{22} & r_{23} & (r_{21}v_x + r_{22}v_y + r_{23}v_z) \\ r_{31} & r_{32} & r_{33} & (r_{31}v_x + r_{32}v_y + r_{33}v_z) \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{15}$$

where $(v_x, v_y, v_z)^T$ is the position of the viewpoint in the world coordinate system.

The OpenGL projection matrix is

$$\mathbf{M}_{\text{PROJECTION}} = \begin{pmatrix} \frac{-2a}{W} & 0 & 1-\frac{2c_x}{W} & 0 \\ 0 & \frac{-2b}{H} & 1-\frac{2c_y}{H} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad (16)$$

where W and H are the width and height of the viewport in pixels, respectively, and n and f are the distances of the near and far plane from the viewpoint, respectively. If n and f cannot be known beforehand, a simple and efficient way to compute good values for n and f is to transform the bounding sphere of the 3D object into the eye coordinate system and compute

$$\begin{aligned} n &= -o_z - r, \\ f &= -o_z + r, \end{aligned} \quad (17)$$

where o_z is the z-coordinate of the center of the sphere in the eye coordinate system, and r is the radius of the sphere.

5 RESULTS

In Figure 4, we show three example results. The images in the leftmost column were generated using symmetric perspective view frusta enclosing the bounding spheres of the respective objects. The middle column shows the bounding quadrilaterals computed using our algorithm described in Section 3. The rightmost column shows the images generated using the new frusta computed using our method. Note that each object is always viewed from the same viewpoint for both the unoptimized and optimized view frusta.

6 DISCUSSION

In this section, we discuss some issues regarding our method.

If the viewpoint is dynamic, a new view frustum has to be computed for every rendered frame. In the computation of the 2D convex hull and the bounding quadrilateral, if the number of 2D image points is too large, it may be difficult to render at interactive rates. For a static 3D object, we can first pre-compute

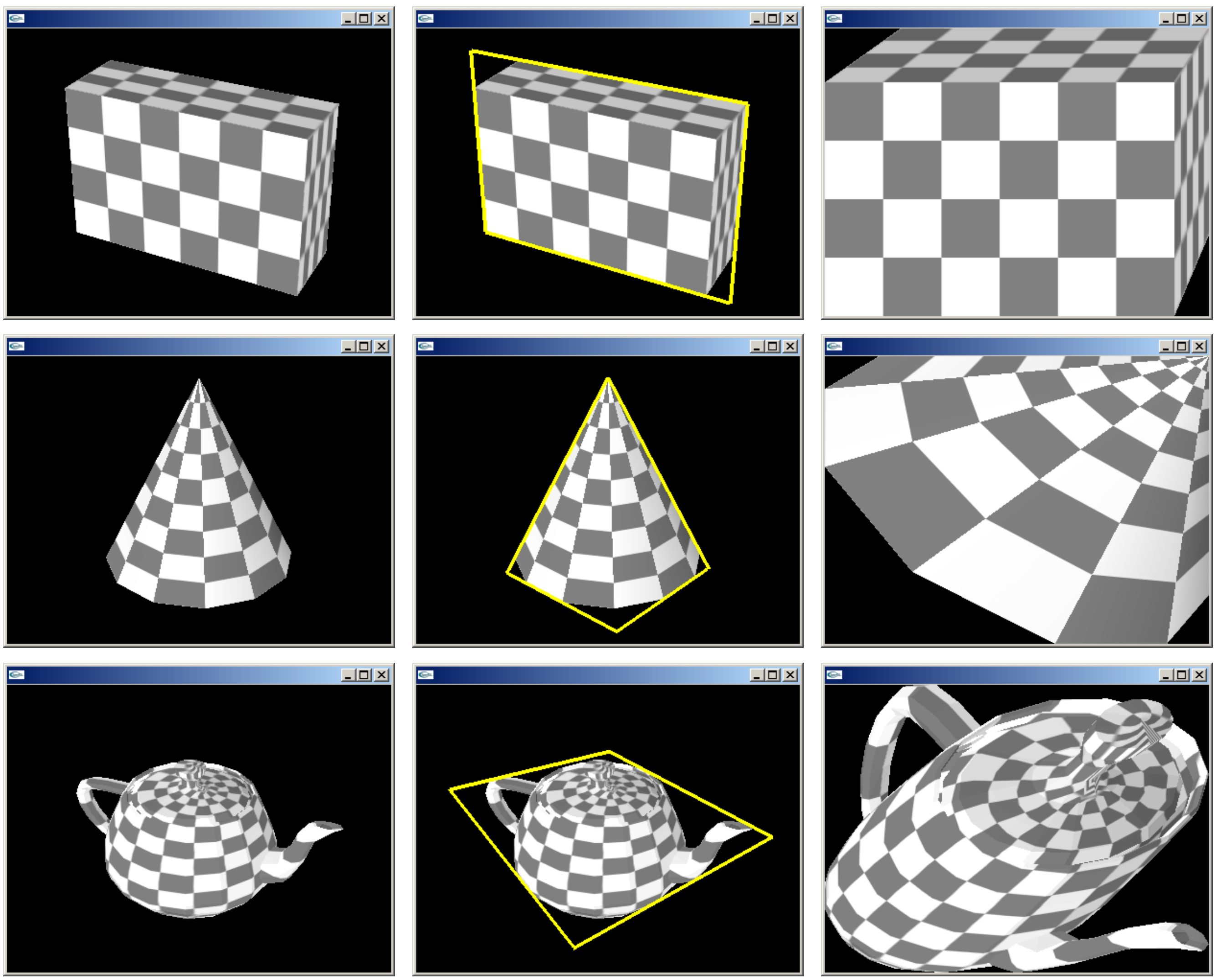

***Figure 4**: Example results.*

its 3D convex hull, and project only the 3D vertices of the convex hull onto the window coordinate system as 2D image points. This will generally reduce the number of 2D points that our algorithm needs to work with. If the 3D convex hull is still too complex, we can simplify it to reduce its number of faces and vertices. Note that the simplified hull should totally contain the original convex hull. The 3D convex hull and its simplified version would be computed in a pre-processing step.

Besides the advantage of increasing the resolution of the object's image, our method can also improve the temporal consistency of the object's image resolution from frame to frame. If the 3D object has a predominantly large face (or a predominant silhouette), the image plane of the computed view frustum will tend to be oriented with it for many viewpoints. This results in a more stable image plane, and therefore more consistent object's image resolution. This benefit is important to projector-based displays in which projective texture mapping is used to produce perspective-correct imagery for the tracked users [Raskar98]. In this application, texture maps are generated from the user's viewpoint, and are then texture-mapped onto the display surfaces using projective texture mapping. Excessive changes in texture map resolution when the viewpoint moves can cause undesired effects in the projected imagery.

Something we wish we had done is to prove how much worse our approximated smallest enclosing quadrilaterals are, compared to the truly optimal ones. Such a proof would most likely be nontrivial. Since we also did not have an implementation of the algorithm described in [Aggarwal85] available to us, we could not do any empirical comparisons between our approximations and the true minimum areas. However, from manual inspection of our results, our algorithm always produced results that are within our expectation of being good approximations of the smallest possible quadrilaterals. Note that even if the quadrilateral is the smallest possible, it still cannot guarantee that the object's image area will be the largest possible. This is because the projective warp does not "scale" every part of the quadrilateral uniformly.

Raskar described a method to append a matrix that represents a 2D collineation to an OpenGL projection matrix to achieve the desired projective warp of the original image [Raskar99]. Though such a 2D projective warp preserves collinearity in the 2D image plane, it does not preserve collinearity in the 3D NDC. This results in incorrect depth interpolation, and therefore, incorrect interpolation of surface attributes. Our method can also be used for oblique projector rendering on planar surfaces. In this case, we usually need to compute the view frustum that warps the rectangular viewport to a smaller quadrilateral inside the viewport. The results from our method do not have the incorrect depth interpolation problem.

7 CONCLUSION

We have demonstrated a simple method to compute an efficient view frustum for a 3D object viewed from a given viewpoint, such that the final object's image area is near-maximal.

The recognition of the 2D relationship between object images in different image planes, and also the relationship between these image planes and their view frusta has allowed us to efficiently perform the optimization in 2D, and then transform the result into a valid 3D frustum.

For the 2D optimization, we have introduced a novel and efficient algorithm to compute a near-optimal tight bounding quadrilateral enclosing a set of 2D points. We have also introduced the use of a well-studied camera calibration technique in computer vision to derive the desired view frustum.

ACKNOWLEDGEMENTS

We wish to thank Jack Snoeyink for referring us to the previous work on minimal enclosing polygons, as well as to Anselmo Lastra and Greg Welch for their many useful suggestions.

Support for this research comes from NSF ITR grant "Electronic Books for the Tele-immersion Age" and NSF Cooperative Agreement no. ASC-8920219: "NSF Science and Technology Center for Computer Graphics and Scientific Visualization."

REFERENCES

[Aggarwal85] Alok Aggarwal, J. S. Chang, Chee K. Yap. *Minimum Area Circumscribing Polygons*. The Visual Computer: International Journal of Graphics, 1:112–117, 1985.

[Berg97] Mark de Berg, Marc van Kreveld, Mark Overmars Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[Faugeras93] Olivier Faugeras. *Three-Dimensional Computer Vision*. MIT Press, 1993.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition*. Addison Wesley, 1990.

[Hoff98] Kenneth E. Hoff. *Understanding Projective Textures*. http://www.cs.unc.edu/~hoff/techrep/projtextures.html, 1998.

[Press93] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, January 1993.

[Raskar98] Ramesh Raskar, Matt Cutts, Greg Welch, Wolfgang Stuerzlinger. *Efficient Image Generation for Multiprojector and Multisurface Display Surfaces*. Rendering Techniques '98, proceedings of the 9th Eurographics Workshop on Rendering, June 1998.

[Raskar99] Ramesh Raskar. *Oblique Projector Rendering on Planar Surfaces for a Tracked User*. SIGGRAPH Sketch, 1999. http://www.cs.unc.edu/~raskar/Oblique/oblique.pdf

[Segal92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, Paul E. Haeberli. *Fast Shadows and Lighting Effects Using Texture Mapping*. SIGGRAPH 92 Conference Proceedings, Annual Conference Series, ACM SIGGRAPH, Addison Wesley, vol. 26, pp. 249–252, July 1992.

[Strang88] Gilbert Strang. *Linear Algebra and Its Applications, Third Edition* (1988). International Thomson Publishing.

[Trucco98] Emanuele Trucco and Allessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

[Williams78] Lance Williams. *Casting Curved Shadows on Curved Surfaces*. Proceedings of SIGGRAPH 78. In Computer Graphics, 12(3):270–274, ACM SIGGRAPH, August 1978.

[Woo99] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner (OpenGL Architecture Review Board). *OpenGL Programming Guide, Third Edition: The Official Guide to Learning OpenGL, Version 1.2*. Addison Wesley, 1999.

APPENDIX

Proving that edge elimination is always possible if $n > 4$

An edge i is a candidate for removal if the sum of the two interior angles it makes with the two adjacent edges is *greater than* 180°. We want to prove that for any n-sided convex polygon, where $n > 4$, there always exists at least one edge that can be removed.

Suppose that for every edge, the sum of the two interior angles it makes with the two adjacent edges is *less than or equal to* 180°. Then the sum of all the interior angles of the convex polygon is less than or equal to $180°n / 2$ (the division by 2 accounts for the double-counting of each interior angle), which is impossible for $n > 4$ because we know that the sum of all the interior angles of the convex polygon should be $180°(n - 2)$. This means that there exists at least an edge such that the sum of the two interior angles it makes with the two adjacent edges is *greater than* 180°.