Fast Non-Linear Projections using Graphics Hardware

Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier, Bernard Péroche

Fast Non-Linear Projections using Graphics Hardware

Jean-Dominique Gascuel[*] Nicolas Holzschuch[†] Gabriel Fournier Bernard Péroche[‡]

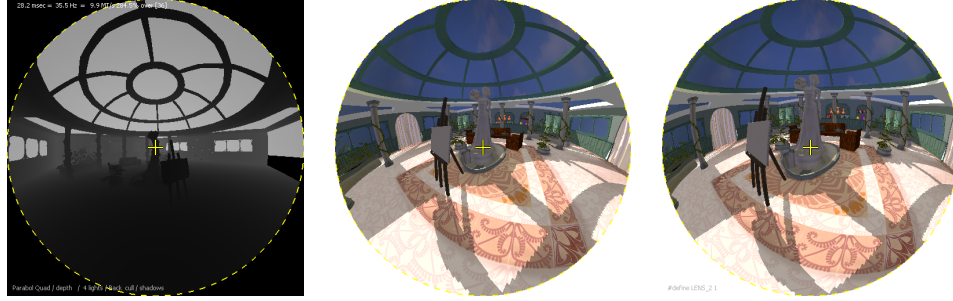CNRS Cornell University / INRIA Dassault Systèmes Université Lyon 1

Figure 1: *Examples of non-linear projections: a paraboloid projection shadow map (left), a cosine-sphere projection (center), and a photographic fish-eye lens used for direct rendering of the scene (right).*

Abstract

Linear perspective projections are used extensively in graphics. They provide a non-distorted view, with simple computations that map easily to hardware. Non-linear projections, such as the view given by a fish-eye lens are also used, either for artistic reasons or in order to provide a larger field of view, *e.g.* to approximate environment reflections or omnidirectional shadow maps. As the computations related to non-linear projections are more involved, they are harder to implement, especially in hardware, and have found little use so far in practical applications. In this paper, we apply existing methods for non-linear projections [Lloyd et al. 2006; Hou et al. 2006; Fournier 2005] to a specific class: non-linear projections with a single center of projection, radial symmetry and convexity. This class includes, but is not limited to, paraboloid projections, hemispherical projections and fish-eye lenses. We show that, for this class, the projection of a 3D triangle is a single curved triangle, and we give a mathematical analysis of the curved edges of the triangle; this analysis allows us to reduce the computations involved, and to provide a faster implementation. The overhead for non-linearity is bearable and can be balanced with the fact that a single non-linear projection can replaces as many as five linear projections (in a hemicube), with less discontinuities and a smaller memory cost, thus making non-linear projections a practical alternative.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation— [I.3.7]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: graphics hardware, non-linear projections, indirect lighting, video games & GPU

---
[*]e-mail: Jean-Dominique.Gascuel@inrialpes.fr

[†]e-mail: Nicolas.Holzschuch@inrialpes.fr

[‡]Université de Lyon, F-69000, Lyon, France; Université Lyon 1, F-69622, Villeurbanne, France; CNRS UMR 5205 LIRIS, F-69622 Villeurbanne, France.

1 Introduction

Most graphics software works with a three dimensional representation of a virtual world and use a two-dimensional device to display it. The *projection* step does the conversion from the 3D representation to a 2D display. It is an essential step of the graphics pipeline. Usually, this projection step is done using linear transforms (in homogeneous coordinates). These transforms are very easy to code, and transform lines into lines and triangles into triangles. But non-linear projections are also used, either to give a larger field-of-view (*e.g.* for environment maps or for ominidirectional shadow maps), or for artistic reasons. In the absence of a fast algorithm for computing non-linear projections, programmers resorted to either conversion from a cube map, or software rasterization.

In this paper, we adapt existing methods for non-linear projections [Lloyd et al. 2006; Hou et al. 2006; Fournier 2005] to a specific class of projections. By restricting ourselves to projections with a single center of projection, radial symmetry and convexity, we get the useful property that each triangle in 3D is converted to a single curved triangle in screen space, making the whole process easier. We provide mathematical analysis for several projections inside this class, such as the paraboloid projection, the cosine-sphere projection, Lambert conformal projection and two fish-eye lenses.

We have found that non-linear projections are, understandably, slower than their linear counterparts. However, for the projections we have studied, the extra cost remains manageable. As a single non-linear projection can be used to replace up to 5 renderings (in a hemicube), it can even be a practical alternative.

Our paper is organized as follows: in the next session, we review previous work on the computation and use of non-linear projections. In section 3, we present our algorithm for computing non-linear projections. In section 4, we review several practical non-linear projections along with their mathematical properties. In section 5, we conduct an experimental study our algorithm. Finally, in section 6, we conclude and present directions for future work.

2   Previous Work

Non-linear projections predate Computer Graphics. In the field of mathematics, Lambert [1772] designed several non-linear projections, mainly for use in cartography. He designed them so that they maintain an important property, such as area preservation or angle preservation. Fish-eye lenses, used in photography to present a very large field-of-view (up to π), represent another example of non-linear projections [Kumler and Bauer 2000].

Inside Computer Graphics, the cosine-sphere projection is often used in lighting computations [Kautz et al. 2004], as the area covered by the objects on the screen is proportional to the light they are reflecting toward to the sampling point. The paraboloid projection, introduced by Heidrich and Seidel [1998] has two advantages: first, it is mapping a large field-of-view (π) with minimal area distortions, and second it is possible to retrieve information using linear tools.

However, *computing* an actual non-linear projection is a difficult problem. In recent years, Kautz *et al.* [2004] used an optimized software rasterizer to compute local occlusion (using the cosine-sphere projection method). Brabec *et al.* [2002] used the paraboloid projection of [Heidrich and Seidel 1998] for omnidirectional shadow maps. Osman *et al.* [2006] showed how this projection can be used in practice for video games, and Laine *et al.* [2007] used the same projection for fast computations of indirect lighting. As they could not compute a complete non-linear projection, they did so by computing the correct projection for all the vertices, followed by a linear interpolation. This method forces them to tesselate the scene into small triangles, at the expense of computation time.

Hou *et al.* [2006] computed specular reflections on curved surfaces by computing the non-linear projections associated with them. They interpolates between non-linear projections, and computes each projection by enclosing the reflected triangle into an enclosing shape, then discarding the extra fragments. The main drawback is that they render every scene triangle for every camera, resulting in a large number of triangles rendered. Lloyd *et al.* [2006; 2007] introduced a logarithmic non-linear projection method for optimal sampling in shadow maps and present different methods for practical rasterization on the GPU, including enclosing each projected triangle into an enclosing shape, then discarding the extra fragments.

Our work shares some similarities with previous work[Lloyd et al. 2006; Hou et al. 2006; Fournier 2005], including the use of an enclosing shape for non-linear projection. Compared to these previous work, the specific of our work is that by restricting ourselves to a specific class of non-linear projections with a single center of projection, we are able to provide a tighter bounding shape. Especially, the projection of a triangle is always a single curved triangle, and we render each scene triangle only once. We offer a comprehensive mathematical analysis of the projection methods, giving the equation of a curved edge, providing tighter bounds.

3   The algorithm

3.1   Overview of the Algorithm

The core of our algorithm is identical to previous work [Lloyd et al. 2006; Hou et al. 2006; Fournier 2005] (see Fig. 2 for pseudo-code); we treat each primitive separately. Each graphics primitive (usually a triangle) will be projected into a non-linear shape (a curved triangle). We first compute a bounding shape for the projection of the primitive, then for each fragment in the bounding shape, we back-project it to the plane of the original primitive, and test whether it

```
for each graphics primitive in 3D {
  compute bounding shape in screen space
  rasterize bounding shape
  for each pixel in bounding shape {
    compute direction in 3D
    intersect ray with 3D primitive
    if (intersection) {
      interpolate depth, color...
      output fragment
    } else {
      discard fragment
    }
  }
}
```

Figure 2: *The algorithm for non-linear rasterization.*

is inside or outside the original primitive using a ray-triangle intersection method.

Fragments that are back-projected outside the primitive are simply discarded. Fragments that fall inside the original primitive are kept. The ray-triangle intersection gives us the barycentric coordinates of the 3D point corresponding to the current fragment. We use these to interpolate the properties of the fragment, such as depth, normal, color, texture... This interpolation is done over the original triangle, using barycentric coordinates. We have to do the interpolation in the original triangle, before projection, because of the non-linear nature of the projection. Even the depth of the fragment has to be interpolated on the triangle instead of in screen space. For efficiency, we only interpolate the useful values: *e.g.* only the depth for a shadow map.

There are two main steps in the algorithm: computing an efficient bounding shape, and testing each fragment for rejection. The former is done by the geometry engine. We have designed two methods for this task: one using triangles (section 3.3), the other using quads (section 3.4). The former offers tighter spatial bounds, while the latter has a smaller geometric complexity. Testing fragments for rejection is done in the fragment engine, and uses a ray-triangle intersection method (section 3.5) [Möller and Trumbore 1997].

In section 3.2, we review the generic properties of our non-linear projections. We prove that for each of them, the projection of a triangle is a curved triangle.

3.2   Non-linear Projections

We are working with specific non-linear projections, namely projections with a single center of projection, depending only on the *direction* to the projected point, with radial symmetry. As a consequence, they are all defined by an equation in the form $r = f(\theta)$. From a direction in 3D, expressed in spherical coordinates (θ, ϕ), we get a position in screen space expressed in polar coordinates (r, ϕ). We add the restriction that the function f must be convex. Table 1 lists interesting non-linear projections we have identified that fall in this class, along with their equations.

As all our projections being defined by $r = f(\theta)$, we can visualize them as a surfaces of revolution S, of equation:

$$z = \frac{r}{\tan\theta} = \frac{f(\theta)}{\tan\theta} \tag{1}$$

Computing the projection of a point M is equivalent to finding the intersection between S and the (OM) line, then projecting this intersection to the projection plane (see Fig. 4(d)).
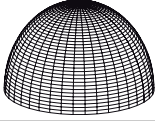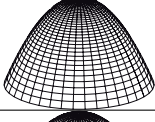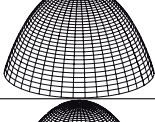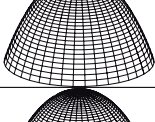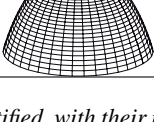
Name	Equation	Surface	Properties
Cos-Sphere	$r = \sin\theta$		Useful for indirect lighting: the screen size of the objects is proportional to their solid angle times the cosine of the angle with the normal. Combined with their radiance, stored in each pixel, we get the incoming indirect lighting from the picture created
Paraboloid	$r = \frac{1-\cos\theta}{\sin\theta}$		Projects a half-universe onto a disc with minimal area and angular distortions.
Fish-Eye (1)	$r = \sqrt{2}\sin\frac{\theta}{2}$		Fish-eye lenses are used in photography to give a view of the half-universe, with some distortions. This form of fish-eye lenses is the most common in fish-eye lenses.
Fish-Eye (2)	$r = \frac{2}{\pi}\theta$		This form of fish-eye lenses is regarded by some as better than the previous, but is very hard to make in camera lenses.
Lambert	$r = \sqrt{1-\cos\theta}$		A Equal-area projection: the screen space area of objects is proportional to the solid angle they subtend, making this the ideal projection for ambient occlusion fields.

Table 1: *The non-linear projections we have identified, with their mathematical definition and properties. Each projection transforms a point in spherical coordinates (ρ, θ, ϕ) into a point expressed in polar coordinates $(r = f(\theta), \phi)$.*

We consider an triangle in 3D space, $[ABC]$, and are interested in its projection in screen space. The projection of each line (AB) is simply the intersection between the plane (OAB) and the surface S. As the function f is convex, this intersection is in a single piece. Thus, each line in 3D space projects to a single continuous curve in screen space. As a consequence, each triangle in 3D space projects to a single curved triangle in screen space.
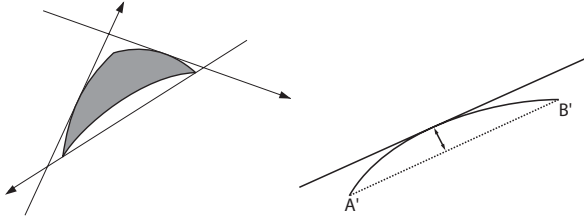
3.3 Triangle bounding shape

Figure 3: *(left) Enclosing a curved triangle with three lines, (right) We bound each edge using a line tangent to the curved edge.*

In this section, we compute a triangle-based bounding shape for the projection of a 3D triangle. We treat each edge of the 3D triangle separately. For each edge, we generate an oriented line in screen space, such that the curved edge is included into its half-space. Combining these three lines gives us a triangle in projection space, that encloses the projection of the triangle (see Fig. 3).

Most of the time, this method does not change the number of vertices in the primitive: a triangle is converted into a triangle. If the triangle is clipped by the projection boundary, however, it can result in two triangles being generated (see Fig. 4(a) and (b)). And for some elongated triangles, the non-linear projection results in angles larger than π (see Fig. 4(c)); such triangles have to be subdivided, and each of the new triangles can also be clipped, resulting in up to 4 triangles being generated.

3.3.1 Bounding each edge

We start with an edge $[AB]$ of the original primitive. We know the projection of its endpoints, A' and B', as well as the linear edge $[A'B']$ connecting them in projection space. This linear edge is not equal to the projection of $[AB]$, the curved edge $[\widehat{A'B'}]$, but the two are connected at the endpoints.

The distance between them is therefore null at A' and B', so it must have at least one extremum between A' and B'. For our non-linear projections, using mathematical analysis (section 4), we can prove that this extremum is unique, and compute its position.

We then check the respective positions of this extremum and the triangle with respect to the line $[A'B']$. If the extremum and the triangle are on the same side of the line, then $[A'B']$ is an acceptable bounding line for this edge. Otherwise, the line parallel to $[A'B']$ passing through this maximum point is the bounding line. As we used lines parallel to the original lines, the resulting set of lines gives us a triangle in projection space, that encloses the projected triangle (see Fig. 3).

3.3.2 Clipping by the projection boundary

We have to do a specific treatment when the original triangle is clipped by the projection boundary:

- When two vertices are clipped, we are left with a curved triangle, whose edges are the clipped edges and the boundary of the projection space (see Fig. 4(b)). We treat this triangle as a normal triangle.

- When a single vertex is clipped (see Fig. 4(a)), we have a quadrangle in 3D space, the projection boundary being the extra edge. This corresponds to 2 triangles. We treat each of these triangles separately, thus doubling the number of vertices.
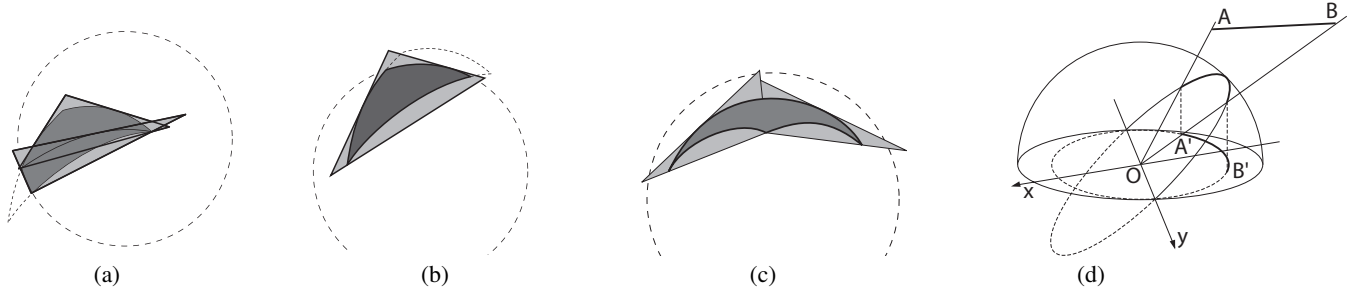
Figure 4: *In some circumstances, we need to add extra vertices to the enclosing shape. (a) one of the vertices is clipped by the projection boundary, creating 2 triangles in 3D, so it is replaced by two larger 2D triangles, resulting in 3 extra vertices. (b) two vertices are clipped by the projection boundary, we keep a single triangle. (c) after projection, some angles are larger than π. We need to subdivide these, resulting in two triangles (which can also be clipped). Hence the maximum output is four triangle for each input triangle. (d) The projections we have identified can be expressed as surfaces in 3D. Projecting an edge is equivalent to finding the intersection between this surface and a plane, then projecting this intersection on the projection plane. Here, the cosine-sphere projection, where the surface is a hemisphere, and x, y the axis of a supporting ellipse.*

3.3.3 Angles larger than π

In some circumstances, the projection of a triangle is a curved triangle where the angle between the edges is larger than π (see Fig. 4(c)). These usually corresponds to thin triangles that are close to the horizon. This situation makes it impossible for our three bounding lines to connect into a triangle that contains the curved triangle. When we detect that case, we subdivide the original triangle, and bound each triangle separately. This case can not happen on any of the triangles after the subdivision, because we subdivide at the problematic angle: the angle at the new triangles is half the angle at the original triangle, thus smaller than π.

3.3.4 Analysis

In most circumstances, this method replaces a triangle with another triangle. When the curved triangle is close to a linear triangle, the bounding triangle will be close to the curved triangle, resulting in a smaller overdraw. However, in the worst case, a triangle can be clipped by the projection boundary, then each of the triangles will be split because of an angle larger than π after projection, resulting in 4 triangles being output for a single 3D triangle. As the speed of the geometry engine depends on the maximum number of triangles that the shader can output, this worst case is slowing down the triangle-bounding method significantly.

3.4 Quad bounding shape

We now bound the curved triangle by a quad. We start by treating each edge of the 3D triangle separately. For each edge, we compute a bounding quad that encloses the curved edge. We then compute a bounding box or a bounding quad for these bounding elements. This method results in a larger overdraw, but it is also more robust, as it does not require line intersections. Since it consistently outputs a single quad for each triangle, the geometry pass is roughly three times faster than with the previous method.
For each edge: we compute a bounding quad, using the geometric properties of the curved edge (obtained with a mathematical analysis, see section 4).
Enclosing the triangle: is trivially done by union of their axis-aligned bounding box. Other methods provide a tighter fit, at the expense of computation time.

3.5 Fragment testing and interpolation

The main step in our algorithm is testing whether each fragment actually belongs in the projected primitive; this step is identical to previous work [Lloyd et al. 2006; Hou et al. 2006; Fournier 2005]. Our input is the coordinates of the current fragment, and information about the original primitive in 3D: the vertices and the normal to the plane. We first convert the fragment coordinates into a ray direction in 3D space. As we have screen coordinates, it requires computing the third coordinate, which depends on the projection method.

This direction, combined with the projection center, defines a ray in 3D space. We test this ray for intersection with the original primitive. We use for this the algorithm by Möller and Trumbore [1997], as we have found it to be faster than other algorithms on modern GPUs.

If there is an intersection, this algorithm also gives us the barycentric coordinates of the intersection point. We use these to interpolate the coordinates and values for the point: depth, color, normal, texture coordinates. Because of the non-linearity of the projection method, none of these can be linearly interpolated in screen space. We output these values, including depth in the z coordinate of the fragment.

3.6 Implementation details

Computing the enclosing primitive for each 3D primitive is done in the geometry engine. We take as input the coordinates of the vertices, and output the 2D coordinates, in screen space, of the enclosing primitive. The geometry shader outputs the vertices of the enclosing shape in screen space. All the information about the original primitive is output as constant parameters for this shape: vertices, normal to the plane and characteristics such as color, normals and texture coordinates. Testing whether a given fragment belongs to the actual projection, as well as interpolating the values for each fragment, is done in the fragment shader.

4 Mathematical Analysis of specific non-linear projections

In this section, we conduct a mathematical analysis of specific non-linear projections (see Table 1), and give the equation of the projection of a linear edge. All our projections are defined by $r = f(\theta)$,
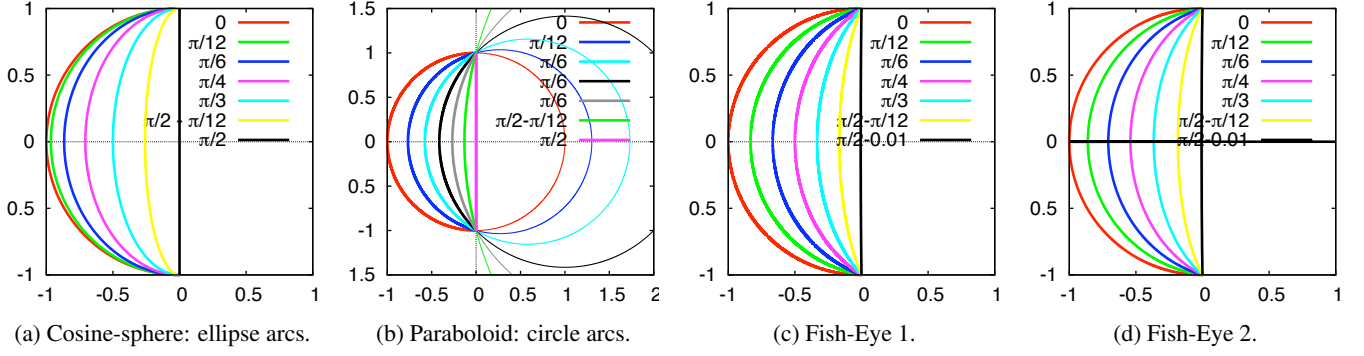
Figure 5: *Non-linear projections convert straight lines in 3D into curves in screen space.*

and we can visualize them as a surfaces of revolution S (see equation 1).

Computing the projection of a point M is equivalent to finding the intersection between this surface S and the line (OM), then projecting this intersection to the projection plane. In all this section, we consider an edge in 3D space, $[AB]$, and we are interested in its projection in screen space (see Fig. 4(d)). The projection of the supporting line (AB) is the intersection between the plane (OAB) and the surface S. Noting $\boldsymbol{n}$ the normal to the plane (OAB), without loss of generality we can orient the coordinate system in screen space so that the x axis is aligned with $\boldsymbol{n}$. $\boldsymbol{n}$ is defined by its angle with the z axis, α, and the equation of the plane (OAB) is:

$$x \sin \alpha + z \cos \alpha = 0 \qquad (2)$$

The intersection of the plane and the projection surface is:

$$x = -\frac{r}{\tan \theta \tan \alpha} \qquad (3)$$

In screen space, we have (by definition of r):

$$x^2 + y^2 = r^2 = f(\theta)^2 \qquad (4)$$

4.1 Cosine-Sphere Projection

For the cosine-sphere projection, we have $r = \sin \theta$, and thus:

$$
\begin{aligned}
x &= -\frac{\cos \theta}{\tan \alpha} \\
\cos \theta &= -x \tan \alpha \\
r^2 &= 1 - x^2 \tan^2 \alpha
\end{aligned}
$$

Plugging this into equation 4, we get:

$$\frac{x^2}{\cos^2 \alpha} + y^2 = 1 \qquad (5)$$

Thus the projection of a line is an ellipse arc (see Fig. 5(a)). The ellipse is centered on the center of projection O, whose great axis is the intersection of the plane (OAB) with the projection plane. This ellipse is the projection of a circle in the plane (OAB) (the intersection between this plane and the hemisphere). This information helps in finding enclosing shapes.

- **for an enclosing line:** instead of considering the ellipse arc in projection space, we consider the circular arc defined by the

intersection of the plane (OAB) and the sphere S. Computing the tangent to that circular arc at any point is easy, through a rotation of $\pi/2$ in the plane. We use the middle of the circle arc, then take its tangent in the (OAB) plane. The projection of the tangent to the circle is the tangent to the ellipse.

- **for an enclosing quad:** we have the projections of the end point of the ellipse arc, and the axis of the ellipse. We use these to create a bounding quad.

4.2 Paraboloid Projection

For the paraboloid projection, we have:

$$
\begin{aligned}
r &= \frac{1 - \cos \theta}{\sin \theta} \\
r^2 &= \frac{1 - \cos \theta}{1 + \cos \theta} \\
x \tan \alpha &= -\frac{r}{\tan \theta} = -\frac{\cos \theta}{1 + \cos \theta} \\
r^2 + x \tan \alpha &= \frac{1}{1 + \cos \theta} = 1 + x \tan \alpha \\
r^2 &= 1 - 2x \tan \alpha
\end{aligned}
$$

Plugging this last result into equation 4, we get:

$$
\begin{aligned}
x^2 + y^2 &= 1 - 2x \tan \alpha \\
(x + \tan \alpha)^2 + y^2 &= 1 + \tan^2 \alpha \\
(x + \tan \alpha)^2 + y^2 &= \frac{1}{\cos^2 \alpha} \qquad (6)
\end{aligned}
$$

Thus the projection of a line is a circle arc (see Fig. 5(b)). The circle is centered on the x axis (defined by $\boldsymbol{n}$), at a distance $-\tan \alpha$ of O, with a radius equal to $1/\cos \alpha$. This information makes it easy to find enclosing shapes:

- **for an enclosing line:** we know the endpoints of the arc, and the center of the circle. We can easily find the middle of the arc, C, and the tangent to the circle at C is an enclosing line.

- **for an enclosing quad:** as we know the endpoints of the arc and the center of the circle, it is easy to find an enclosing quad.

4.3 Fish-Eye Lenses (1)

For the first type of fish-eye lenses, we have $r = \sqrt{2} \sin(\theta/2)$. Obtaining an equation for the curves generated is tricky. Using
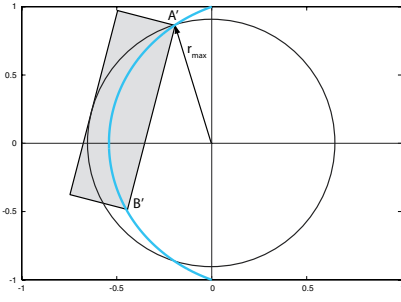
$$x \tan \alpha = -\frac{r}{\tan \theta}$$

Figure 6: *Generic bounding strategy: we enclose each curved edge by its secant $[A'B']$ and a circle arc, whose radius is equal to the maximum radius on the edge, then enclose the circle arc by its tangent.*

we obtain easily:

$$x \tan \alpha = \frac{1 - r^2}{\sqrt{2 - r^2}} \tag{7}$$

To invert this equation, we use the change of variables $u = 1 - r^2$ and $a = x \tan \alpha$. We can solve the resulting polynomial in u:

$$u = \frac{a}{2} \left(a - \sqrt{a^2 + 4} \right)$$

Replacing u by $1 - (x^2 + y^2)$, we get the equation of the curve in cartesian coordinates:

$$y^2 = 1 - x^2 - \frac{x \tan \alpha}{2} \left(x \tan \alpha - \sqrt{x^2 \tan^2 \alpha + 4} \right) \tag{8}$$

Which defines a convex curve, symmetric with respect to the x axis. r is maximal for $x = 0$ ($r = 1$), minimal for $y = 0$ (see Fig. 5(c)).

To find an enclosing quad, we do not need to actually compute this equation, though. We first find the projection of the endpoints. The secant (the line joining the endpoints in screen space) gives us one of the sides of the quad. We take two lines orthogonal to the secant for two other sides. For the remaining side, we take the maximum value of r (which is reached at one of the end points, build a circle arc of radius r with the same angular extent, and take a line that encloses this circle (see Fig. 6).

4.4 Fish-Eye Lenses (2)

For the second type of Fish-Eye lenses, we have $r = 2\theta/\pi$. Thus we have easily θ as a function of r, but we didn't find a simple equation for the curve expressed in Cartesian coordinates. However, we can express the equation of the curve in polar coordinates $(r(\phi), \phi)$:

$$\cos \phi = -\frac{1}{\tan \theta \tan \alpha}$$

$$r = -\frac{2}{\pi} \arctan \left(\frac{1}{\cos \phi \tan \alpha} \right)$$

$$r = 1 + \frac{2}{\pi} \arctan \left(\cos \phi \tan \alpha \right)$$

This function $r(\phi)$ defines a convex curve (see Fig. 5(d)), symmetric with respect to the x axis. r has a single minimum, on the x axis, where $r = 1 - \frac{2}{\pi} \alpha$ and two maxima on the y axis, where $r = 1$.

This information makes it possible to find an enclosing quad for each projected edge (see Fig. 6). We know the projection of the
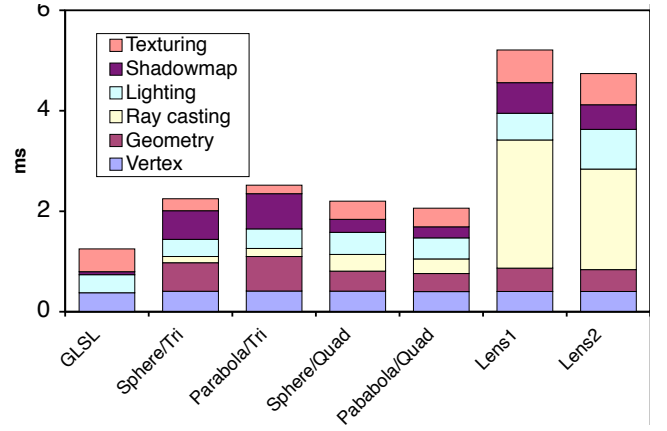

Figure 7: *Time spent in each step of the algorithm, for several projections and bounding methods (*Facade *scene, 2800 triangles).*

endpoints. The secant line forms the basis of our quad, and two lines orthogonal to the secant form the sides. For the last line, we take the maximum value of r in the interval (which is at one of the endpoints), then take a tangent to a circle with this radius. The resulting quad encloses the edge, although it is over-conservative.

4.5 Lambert conformal projection

Although the equation of Lambert's conformal projection looks very different from the equation of the first fish-eye projection, the two surfaces of revolution they give are identical, and the two projections are actually the same (because $1 - \cos \theta = 2 \sin^2 \frac{\theta}{2}$). Thus the strategy designed for the first type of fish-eye lenses will also work for Lambert Conformal projection.

4.6 Generic bounding strategy

Our observation of four sample non-linear projections brings us to devise a generic bounding strategy, that should work for most non-linear projections in our class. We observe that for all the projections we have found, the projection of a line is a convex curve, symmetric with respect to the x axis, with a radius equal to 1 for $x = 0$, and a radius minimal for $y = 0$. Assuming these properties hold for a particular projection method, the bounding strategy used for both Fish-Eye lenses projections will work (see Fig. 6):

- project the endpoints of the edge,

- the line joining these endpoints in screen space provides one side of the bounding quad,

- two lines orthogonal to the secant provide two other sides,

- finally, for bounding the external side of the curve, we take the maximum radius (which is reached at one of the endpoints), build a circle arc with that radius and the same angular extent as the projection of the edge and take the tangent at the middle of this arc.

This method may result in substantial overdraw (the bounding quad is much larger than the projection of the edge).

5 Results and Discussion

All the timings in this section were computed on a Core 2 Duo at 2.4 GHz, and the latest mid range graphic card (Nvidia 8800 GT / 512 MB / driver 169.02).

5.1 Test scenes

We have chosen 4 input scenes, with varying object complexity, ranging from a relatively low number of primitives up to 560K triangles. The first one, *Facade* (see Fig. 10, in the color section), is showing a city street with a small number of textured polygons. By varying the level of detail on the houses, we make the polygon count vary from 600 to 4900 triangles. The second one, *Temple* (see Fig. 8(a), in the color section), consists of 4K triangles. It is our overdraw "worst case", because it consists of columns, made of many thin triangles. The third one, *Characters* (see Fig. 9, in the color section) contains some high-resolution characters on a simple background. The last one, *Patio* (see Fig. 8(b) and (c), in the color section) is a large furnished room, with 560K triangles. It has both large polygons (for the floor, the walls and the roof) and tiny polygons, *e.g.* for the leaves of the plants and the furnitures.

5.2 Cost for each step of the algorithm

Our first goal is to understand how the GPU is working on our algorithm, and to identify the bottlenecks depending on the test scene and the projection. In this section, we are testing a full rendering, with shadow mapping inside the projection and per-pixel lighting. We have run our program in several degraded rendering modes, to identify the cost for each step. For example, to measure the cost of the geometry shader, we run the program without a geometry shader, measure the rendering time and compare with the rendering time with the geometry shader. We did the same thing with no fragments output, no lighting computation, no shadow map computation and not texturing the output. We also measured the total number of fragments accessed by the algorithm (using occlusion queries), and compared it with the number of fragments output, to measure the amount of overdraw caused by our method. We compared these timings with the classical GLSL pipeline, with shadow mapping and per-pixel lighting, but with no geometry shader.

Table 3 and Fig. 7 show our results, for our test scenes and for several projection methods[1]. One important and expected result is that the **geometry** step takes longer for the triangle bounding strategy than for the quad bounding strategy (approximately twice as long). This is due to the worst case scenario for the triangle bounding strategy, which corresponds to four triangles being created, whereas the quad bounding strategy consistently outputs two triangles. The time spent in the **ray-casting** step is related to the amount of overdraw: we have to shoot a ray for each pixel in the bounding shape. In this step, the triangle bounding strategy outperforms the quad bounding strategy, because of a smaller amount of overdraw. The two fish-eye lenses methods perform very poorly for this particular scene, because of the huge amount of overdraw they generate. **Lighting** corresponds to the per-pixel lighting; unsurprisngly, it is more or less constant for all projection methods, including the standard method with per-pixel lighting. Perhaps the more surprising result is that the **shadow mapping** step takes much longer with the triangle bounding strategy than with the quad bounding strategy. We are still investigating this, but we suspect that elongated triangles result in frequent cache misses in the shadow map. Finally, the time in the **texturing** step is also related to the amount of overdraw, because we have to transfer the information needed to compute texture coordinates for each fragment accessed, even if it's going to be discarded. The amount of information transferred is slowing down the fragment shader.

In Table 3, the results for our largest test scene (the *Patio*) show that the cost of the geometry step has increased dramatically. This may

---

[1]A file containing all our results and measurements is also included in the supplemental materials.

	Color (ms)	Depth (ms)	Coverage (ms)
$6 \times 512 \times 512$ cubemap	2590	548	490
$2 \times 512 \times 512$ paraboloids	716	157	121
$6 \times 16 \times 16$ cubemap	8.8	5.2	4.8
$2 \times 16 \times 16$ paraboloids	3.4	2.6	2.4
$5 \times 512 \times 512$ hemicube	1500	375	320
$1 \times 512 \times 512$ paraboloid	375	79	61
$1 \times 512 \times 512$ sphere map	433	71	61
$1 \times 512 \times 512$ Lambert	396	80	67
$5 \times 16 \times 16$ hemicube	7.1	4.3	4.0
$1 \times 16 \times 16$ paraboloid	2.5	1.9	1.6

Table 2: *Comparison between hemicube/cube rendering and non-linear projections (in ms) (*Facade *scene, with 1900 triangles).*

be a limitation for using our algorithm on very large scenes.

5.3 Comparison with hemicube/full cube

One of the key uses of non-linear projections is to render a full view of the virtual world, *e.g.* for environment mapping, for indirect lighting or for shadow maps for omnidirectional light sources. For these applications, the other usual method is to compute a hemicube, if you need a view of the half-universe, or a full cube if you need information for all the directions. The drawback is that the hemicube requires 5 different linear projections, and the full cube, 6.

We have run a comparison between non-linear projections and hemicube or full cube rendering of our test scenes (see Fig. 9 and Table 2). We computed the hemicubes and full cubes using the fastest method (at the time of writing), inside a Frame-Buffer Object. We can compare either a full cube rendering with two non-linear projections, or a hemicube with a single non-linear projection. We have tested computing environment maps (with per-pixel lighting and shadow mapping), shadow maps (just computing the depth of the fragment) and coverage maps (just testing whether the fragment is covered or not).

The main result is that the non-linear projections consistently outperform the hemicube and the cube rendering. For this test scene, it seems that the time required to create each projection and view frustum is slowing down the hemicubes rendering. This result may not hold for larger scenes, if the rendering becomes geometry-limited, but it is important in that it shows that non-linear projections are a practical alternative to hemicube or cube renderings.

5.4 Overdraw

As our method encloses the curved projection of each primitive in a larger one, there is a certain amount of overdraw. Fig. 9 shows the pixels accesed by our algorithm, for the two bounding methods. Red/Green display the barycentric coordinates computed, while blue shows discarded pixels. Several important informations are visible in these pictures: first, for the smaller triangles in each of the hi-resolution characters, the triangle bounding method provides an almost perfect fit, resulting in a low percentage of overdraw, 37 %. Second, the triangle bounding method results in some very elongated triangles, which has a bad effect on caching schemes. Third, the front-most triangle of the ground has been clipped, and replaced by two triangles. Fourth, although the quad bounding method generates a larger overdraw (107 %), it still runs faster on this scene.

	covered (K pix)	rendered (K pix)	overdraw	vertex (ms)	geometry (ms)	raycasting (ms)	lighting (ms)	shadow (ms)	texturing (ms)	Total (ms)	Ratio
GL		249	0%	0.40	-	-	0.33	0.06	0.45	1.25	100%
ST	308	173	78%	0.41	0.57	0.13	0.34	0.57	0.24	2.25	180%
PT	406	197	106%	0.41	0.69	0.16	0.39	0.70	0.17	2.52	202%
SQ	897	173	418%	0.41	0.40	0.33	0.44	0.26	0.36	2.20	176%
PQ	714	181	294%	0.40	0.36	0.29	0.42	0.22	0.37	2.06	165%
Lens1	5089	172	2859%	0.40	0.47	2.55	0.53	0.61	0.65	5.21	417%
Lens2	5006	172	2810%	0.40	0.44	2.00	0.79	0.49	0.62	4.74	379%
600	702	181	288%	0.39	0.16	0.26	0.44	0.17	0.13	1.55	128%
2800	714	181	294%	0.40	0.36	0.29	0.42	0.22	0.37	2.06	165%
4900	786	182	332%	0.45	0.52	0.29	0.43	0.27	0.49	2.44	185%
Temple PQ	1950	250	680%	0.17	0.33	0.77	0.63	0.42	0.57	2.88	461%
Patio PQ	1271	335	279%	7.55	31.47	0.86	0.24	5.93	38.28	79.85	600%

Table 3: *Rendering times for our algorithm, with the cost of the different steps. The first 7 lines are for the Facade scene (2800 triangles), for several projection methods:* **GL** *(standard GLSL rendering with per pixel lighting),* **S***x is Spherical map;* **P***x is Parabola map; x***T** *uses triangles enclosing shape; while x***Q** *uses quad bounding box. The next three lines are for Facade with different scene complexity for the PQ algorithm. The last two lines are for larger scenes.*

6   Conclusion and Future Directions

In this paper, we have presented a robust algorithm for handling specific non-linear projections inside the graphics pipeline. Our algorithm works both for direct display of the non-linear projection, *e.g.* a fish-eye lens inside a video game, or for indirect use, *e.g.* when rendering a shadow map with a paraboloid projection.

As with previous work, we start by bouding the projection of each shape, then discard extra fragments inside the bounding shape. Our contributions are twofold. First: two different methods for bounding the non-linear projections, one based on triangles that is optimal in fragments but requires more work in the geometry engine, the other based on quads that is optimal for the geometry engine but can causes more overdraw. Second: a mathematical analysis of several non-linear projection methods, where we show that some of them have simple expressions, and thus lend themselves to easy bounding through geometric tools.

Although non-linear projections are slower than linear projections, the extra cost is manageable. As a single non-linear projection can replace up to five linear projections (in a hemicube), it can even be a practical alternative, both for rendering time and memory cost.

References

BRABEC, S., ANNEN, T., AND SEIDEL, H.-P. 2002. Shadow mapping for hemispherical and omnidirectional light sources. In *Computer Graphics International*, Springer, 397–408.

FOURNIER, G. 2005. *Caches multiples et cartes programmables pour un calcul progressif et interactif de l'éclairement global*. PhD thesis, Université Lyon 1.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. View-independent environment maps. In *Graphics Hardware '98*.

HOU, X., WEI, L.-Y., SHUM, H.-Y., AND GUO, B. 2006. Real-time multi-perspective rendering on graphics hardware. In *Rendering Techniques 2006: Eurographics Symposium on Rendering*.

KAUTZ, J., LEHTINEN, J., AND AILA, T. 2004. Hemispherical rasterization for self-shadowing of dynamic objects. In *Rendering Techniques 2004: Eurographics Symposium on Rendering 2004*, 179–184.

KUMLER, J. J., AND BAUER, M. L. 2000. Fish-eye lens designs and their relative performance. In *Current Developments in Lens Design and Optical Systems Engineering*, SPIE, 360–369.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J., AND AILA, T. 2007. Incremental instant radiosity for real-time indirect illumination. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, 277–286.

LAMBERT, J. H. 1772. *Anmerkungen und Zusätze zur Entwerfung der Land- und Himmelscharten.*

LLOYD, D. B., GOVINDARAJU, N. K., TUFT, D., MOLNAR, S. E., AND MANOCHA, D. 2006. Practical logarithmic shadow maps. In *Siggraph 2006 Sketches and applications*.

LLOYD, D. B., GOVINDARAJU, N. K., QUAMMEN, C., MOLNAR, S. E., AND MANOCHA, D. 2007. Practical logarithmic rasterization for low-error shadow maps. In *Graphics Hardware 2007*.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1, 21–28.

OSMAN, B., BUKOWSKI, M., AND MCEVOY, C. 2006. Practical implementation of dual paraboloid shadow maps. In *ACM SIGGRAPH Symposium on Videogames*, ACM Press, 103–106.
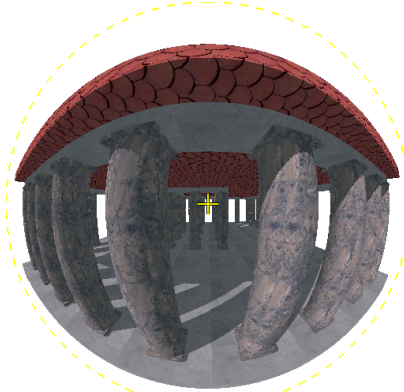
(a) *Temple* test scene (4K triangles, paraboloid projection, 6 ms). The columns generates a lot of overdraw.
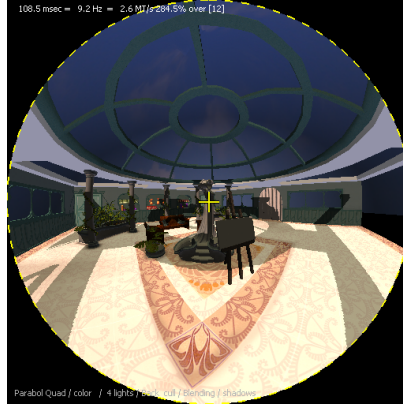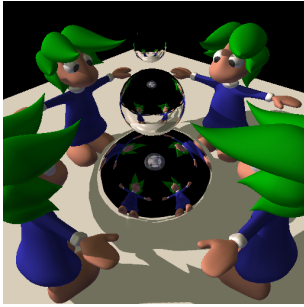
(b) *Patio* test scene (560K triangles, paraboloid projection, 108 ms).
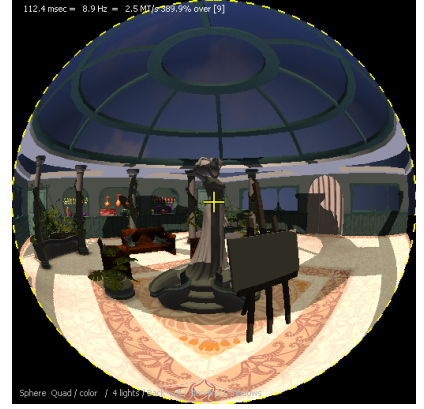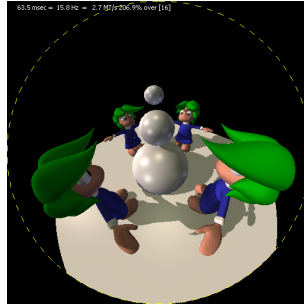
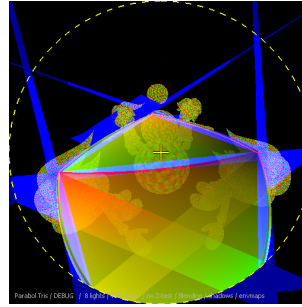(b) *Patio* test scene (560K triangles, spherical projection, 112 ms).

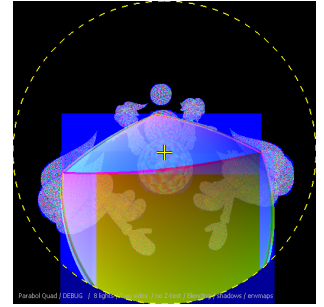**Figure 8:** Temple *and* Patio *test scenes.*

(a) Linear projection.

(b) Paraboloid projection (used for the environment map).

(c) Triangle bounding: 37 % overdraw, 107 ms.

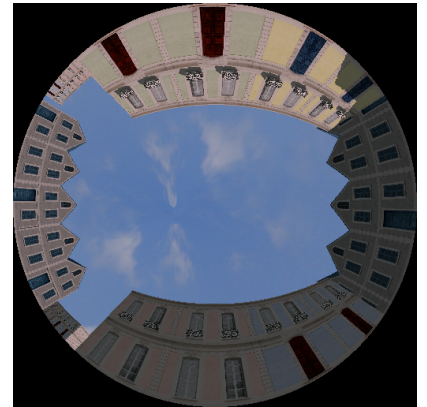(d) Quad bounding: 107 % overdraw, 63 ms.

Figure 9: *The overdraw generated on* Characters *test scene (177 K triangles), with the paraboloid projection.*

(a) Linear projection.

(b) Hemicube view, 512^2 + 4 × (512 × 256) pixels, 1.5 s.

(c) Paraboloid projection, 512^2 pixels, 0.37 s.

Figure 10: *The* Facade *test scene, with hemicube and paraboloid projection (1400 triangles).*