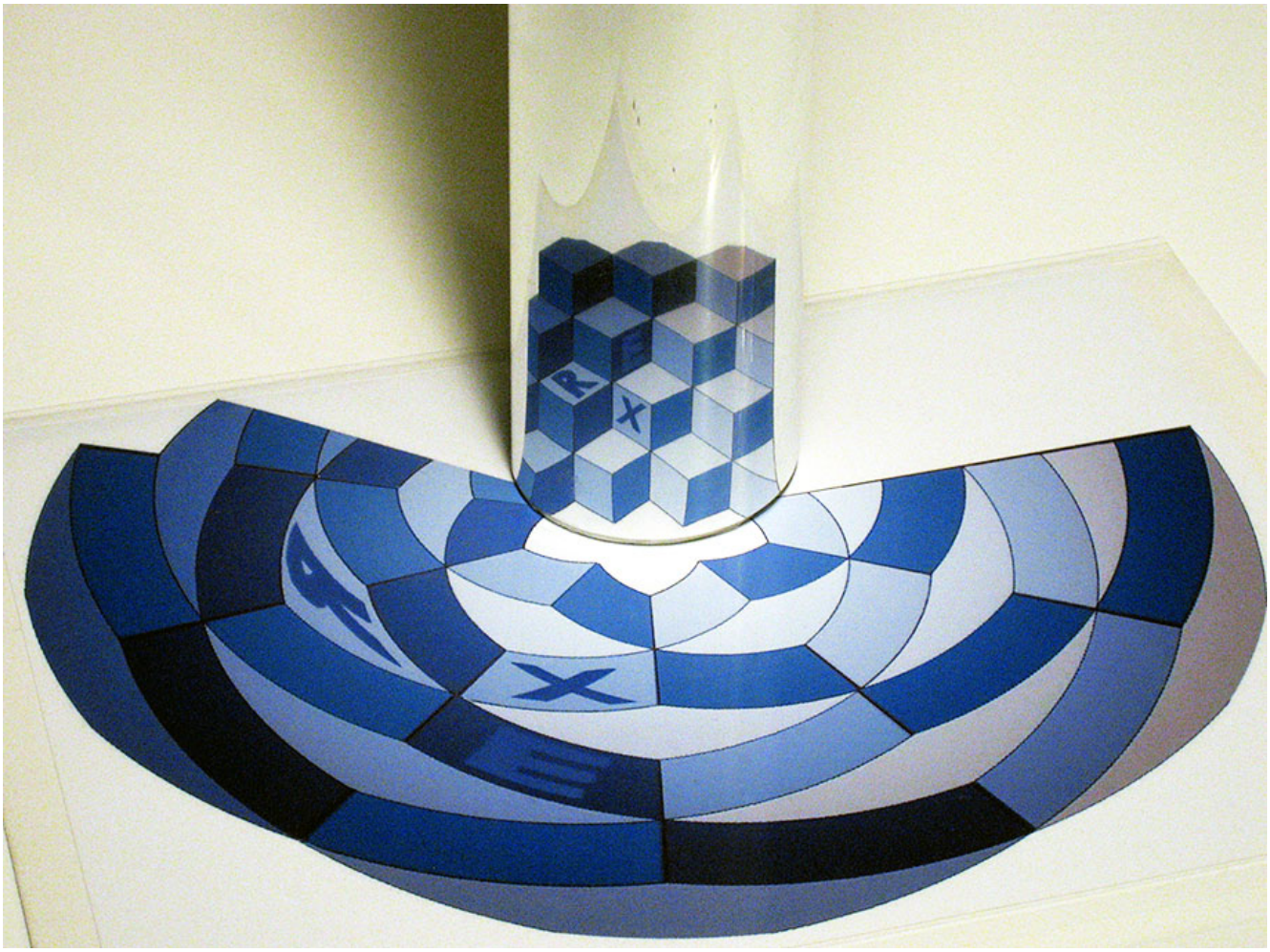
# Raytracing Mirror Anamorphosis

Riley Jense, Baran Usta, Elmar Eisemann

Mirror anamorphosis is characterised by a distorted projection, where the combination of a mirror and a specific viewpoint lets the observer see the undistorted image. The use of this technique and complex perspective in general has gone from being niche to becoming mainstream. Raytracing poses itself as a solution to solving the math for such constructions, as it is capable of delivering accurate geometric calculations and a high degree of visual realism. There is a strong connection between the work needed to construct a mirror anamorphosis, and the computations done in raytracing. Raytracing has been a major topic of research with applications in high-quality image rendering. We propose an algorithm which combines raytracing for mirror anamorphosis with texture mapping. We first generate a set of points based on the desired quality, and triangulate them to a two-dimensional triangle mesh using Delaunay triangulation. Using raytracing, the mesh is projected onto the surface bouncing off the mirror. Surface intersections are recorded in a mapping with its respective texture coordinates. Based on this mesh, we do not have to execute our raytracing algorithm again if the projected image is changed, thus providing us with a significant speedup.



Perspective Artwork using Mirror Anamorphosis (Rex Young)

# 1 Introduction

Mirror anamorphosis is characterised by a distorted projection, where the combination of a mirror and a specific viewpoint lets the observer see the undistorted image. The use of this technique and complex perspective in general has gone from being niche to becoming mainstream. Street art might be the most obvious example for regular perspective art, but the use of mirrors is rather uncommon. A modern day artist making use of mirrors is István Orosz. Their anamorphic art often involves some original image in a mirror, which is projected and heavily distorted on the surrounding surface. Most of the art involves a flat surface and a simple object, like a cone or cylinder, and there is a distinct lack of research on more complex geometry. Generalising calculations for more complex objects and surfaces, together with digitising the method for creating such a construction, would be a major step forward for artists employing these methods.

## 1.1 Raytracing and Textures

Raytracing poses itself as a solution to solving the math for such constructions, as it is capable of delivering accurate geometric calculations and a high degree of visual realism. There is a strong connection between the work needed to construct a mirror anamorphosis, and the computations done in raytracing. Raytracing simulates the interaction between objects and light rays in a virtual environment, whereas mirror anamorphosis is a projection using reflected light. The relation is clear, mirror anamorphosis heavily relies on the behaviour of light, and raytracing can simulate this behaviour. Especially in recent years raytracing technology has seen major leaps and speed improvements, and is a solid candidate for constructing anamorphic projections. Combining raytracing with texture mapping, which is a method to wrap images around objects, allows us to create a transformation from original to distorted image. In recent decades texture mapping has seen multi-pass rendering, multi-texturing and mipmaps, which significantly reduces the amount of calculations needed to construct near-photorealistic images.

## 1.2 Key Concepts

There are several key features to projections like these, many which are already core concepts in computer graphics. The original image will be referred to as the projected image, and is visible in mirror object. The transformed counterpart on the surface is specified as the surface. Combining these aspects, together with a camera and a projection plane, leads to the definition of a scene. The scene and the projected image are passed as input to our algorithm, producing an output consisting of the mapping from source to surface, and the resulting image.

# 2 Related Work

The use of anamorphosis in art is certainly not new. On the contrary, the earliest known example was drawn by Leonardo da Vinci, dating back hundreds of years ago. These are all constructed in the real world, however, and involves a lot of manual labour. Only since the digital age have we been able to create computer generated images on a massive scale. Raytracing has been a major topic of research with applications in high-quality image rendering, visual effects and even real-time video gaming. The combination of anamorphosis and raytracing is not well-studied, with only a limited amount of properly documented methods available.

## 2.1 Anamorphic Methods

Currently there are only a limited amount of sufficiently documented methods available. The most recognised approach is the digitised 'through-hole' method developed by Comité in 2010.[1] The transformation from projected image to surface is based on a set of quadrilaterals using the pixels in the projected image. This grid of pixels becomes distorted as it is projected to the surface, due to the curvature of the mirrors present in anamorphoses like these. Complex shapes where quadrilaterals overlap or are projected onto a non-flat surface, makes it rather tough to determine the appropriate colour for areas on the surface. While a subset of these issues can be mitigated with interpolation, it turns out to be quite costly, much like how raytracing itself can be quite costly for quadrilaterals. Another method is to derive the relevant equations for a certain shape, and compute the resulting transformation. As this is not a general method, the entire process must be repeated for each different mirror shape.[2][3] Furthermore, deriving the equations for objects more complex than cones, cylinders, or similar, turns out to be a strenuous task. While approximating shapes is definitely possible, it is too costly for simulating light-object interaction in raytracing.

## 2.2 Point Selection for Raytracing

Raytracing methods are well-defined, and our basic implementation is provided by Shirley[4]. The framework consists of the architecture for raytracing primitives like rays and camera, important utility functions such as reflection, and simple implementations of common raytracing methods. Among these methods are anti-aliasing, texture mapping and bounding volume hierarchies. Some alternative raytracers are 'pov-ray', used by Comité[1], or 'yet another raytracer', which allows for non-classical perspectives.[5] The raytracer takes as input a set of points from which rays should originate, and a straightforward approach to generating this set is using an equally distributed set of points. As quadrilaterals are translated to triangles in modern GPU architecture[6], we will explore the possibility of using triangles directly. To find a triangulation utilising every point in the given set, we find the

Delaunay triangulation. The Delaunay triangulation also maximises the minimum angle of all the angles in the triangulation. This avoids the inclusion of elongated triangles and provides an optimal mesh to render with.[7][8] S-hull is a excellent algorithm for finding Delaunay triangulations, and is provided by Sinclair.[9]

### 2.3 Reconstruction using Texture Mapping

Texture mapping is a novel technique for high-quality image synthesising, and is analogous to creating distorted images from our mapping based on the original. Apart from determining surface colour, some uses of texture mapping are finding specular reflection, bump mapping, transparency, shadows, and has become a core component in fast rendering.[10] Furthermore, more complex surfaces are made up of triangles as well, and rendering relies heavily on computing intersections between ray and triangle. We use the the 'in-out' test for checking these types of intersections, however, rendering can be sped up by amortising computation over neighbouring triangles. Amanatides & Choi suggest such an approach using Plücker coordinates.[11] Texture mapping goes hand-in-hand with the surface mesh, as it is highly optimised for triangles. This is due to the fast computation of barycentric coordinates.[12]

## 3 Methodology

Overcoming these challenges poses a large roadblock in adding complexity and improving computation speed for projections like these. Not only do we explore fitting approaches for simulating anamorphoses, but also allow the raytracing algorithm we introduce to be optimised specifically for anamorphic art. Moreover, we would want to determine the limitations of various shapes and viewpoints.

In order to start working towards a solution for solving mirror anamorphosis in raytracing there needs to be a skeleton implementation of raytracing with its respective primitives, with a definition of the input and output. The input consists of a projected image, a mirror object, and a surface object onto which we project. The scene is made up of these two objects, together with a camera, from which rays will originate, and a projection plane. The rays are cast through this projection plane, reflecting off the mirror and finally intersecting with the surface. Tracing utilises a function for determining where rays end up within the scene, by checking for intersections with objects. If an intersection occurs, a record of data is generated with the coordinates of the ray-object intersection, together with the normal on the surface of the intersected object. Rays consist of direction and origin, whereas records consist of normal, distance and intersection point, such that

$$d_{ray} = (d_x, d_y, d_z), o_{ray} = (o_x, o_y, o_z) \quad \text{Ray}$$
$$p_{rec} = (p_x, p_y, p_z), n_{rec} = [\, v_1 \;\; v_2 \;\; v_3 \,] \quad \text{Record}$$

### 3.1 Finding Intersections

Using the equation for a given shape of the object, and the ray equation, we can solve the resulting quadratic formula (with discriminant $d$) to determine whether there are intersections, and find the distance $t$ from $o_{ray}$. Furthermore, a lower bound $t_{min}$ and upper bound $t_{max}$ are defined as render distance. Intersection is now given by,

$$d > 0 \quad \text{Solution Exists}$$
$$t_{min} < t < t_{max} \quad \text{Within Bounds}$$

Using a raytracer based on Shirley's implementation[4], we can generate images given a scene description. Using definitions for material and colour, rays are assigned a colour on intersection with an object. Finally, all results are bundled into an output image on a pixel-by-pixel basis. This will be the groundwork for the mirror anamorphosis implementation.

There are various existing solutions using the traditional approach, with one of the most basic scenes consisting of a mirror cylinder and a surface plane. Our approach supports various scenes, but for the sake of simplicity, this elementary scene is set up first. The set-up starts off similar to the through-hole method, with the cylinder as mirror, plane as surface, and an additional plane situated between the viewpoint and the mirror object, as the screen. This screen operates as the location for where in the mirror the source image will be visible, if at all. As an example, moving the screen further away from the mirror, towards the viewpoint, results in a larger image similar to zooming in. Furthermore, moving the screen up to the point rays will not hit the mirror anymore as they pass through the screen will effectively crop the image.

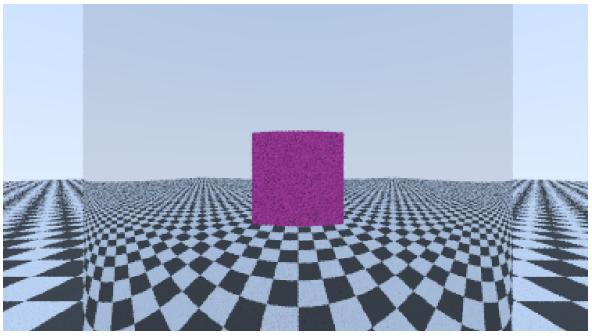
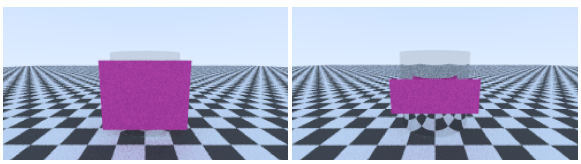


Figure 2: Close-up of cylinder and screen.



(a) Large screen (b) Wide screen

Figure 3: Various screen placements in the scene.

The key part of the implementation is checking whether a ray intersects with both the screen, the mirror, and then hits the surface after reflecting off the mirror. We define some functions, where $normalise(v)$ converts some vector $v$ into its unit vector, and $intersect(r, o)$ tests intersection between ray $r$ and object $o$. Now, determining whether a ray is part of the mirror anamorphosis is given by,

**Algorithm 1** Testing rays for intersection with objects.

**intersection_test** (ray $r$, screen $e$, mirror $m$, surface $s$)
  **if not** $intersect(r, e)$ **then**
    **return** $false$
  **end if**
  **if not** $intersect(r, m)$ **then**
    **return** $false$
  **end if**
  $p \leftarrow$ `intersection point` $(r, m)$
  $n \leftarrow$ `normal at` $m_p$
  $v \leftarrow normalise(r_d)$
  $reflected \leftarrow 2n * dot(v, n)$
  **if not** $intersect(reflected, s)$ **then**
    **return** $false$
  **end if**
  **return** $true$

| | |
|---|---|
| Cylinder | $f(x,y,z) = Ax^2 + By^2 + Cz^2 + Dx + Ey + Gz + H$ |
| Plane | $f(x,y,z) = Ax + By + Cz + D$ |
| Triangle | $f(v_1, v_2, v_3) = v_1 + A(v_2 - v_1) + B(v_3 - v_1)$ |

Table 1: General equations for various shapes.

## 3.2 Selecting Points

Before doing any kind of computation, rays need to have an origin. The naive approach uses a method referred to as the *through-hole method*, by constructing a grid based on the pixels in the source image. This brings forth several limitations with relation to object shape, surface shape and computation speed. To support this claim, let us investigate some differences between these quadrilaterals and triangles. It is given that any three points, when non-collinear, determine a unique triangle which lies on a unique plane. We define the set of triangles to not contain any triangles with a size of zero, as these describe points and lines containing three vertices. Transforming these vertices from image to surface, will almost always result into a new two-dimensional triangle. The same cannot be said for quadrilaterals, and issues arise when the surface is a non-flat object. What further imposes the superiority of triangles is the fact that it is the only primitive that can be described in isolation, due to the aforementioned property. This is exactly the reason many modern systems rely on triangles for rendering, and not quadrilaterals. Lastly, triangles scale linearly, which makes the computation of various interpolations for shading, texturing and applying depth filters highly optimised. A straightforward approach might be splitting every quadrilateral into two triangles. However, with a curved mirror surface points might end up quite far apart, producing a low quality output. Better quality can be achieved by using more points, or rather, use a higher sampling rate. Points are selected in a randomised manner per pixel.

Depending on the quality we want, we use a sampling rate $x$. The amount of points generated is given by $x * pixels$ and is computed through the following procedure, where `random_double` generates a random value in range $[0, 1]$.

**Algorithm 2** Generating points to cast from.

**generate_points** (height $h$, width $w$, samples $x$)
  `Let` $A[1...h][1...w]$ `be a new array`
  **for** $i \leftarrow h$ `to` $0$ **do**
    **for** $j \leftarrow 0$ `to` $w$ **do**
      **for** $k \leftarrow 0$ `to` $x$ **do**
        $u \leftarrow (j +$ `random_double` $) / (w - 1)$
        $v \leftarrow (i +$ `random_double` $) / (h - 1)$
        $A[i][j] \leftarrow (u, v)$
      **end for**
    **end for**
  **end for**
  **return** $A$

## 3.3 Triangulation

Generating a set of non-overlapping edge-adjoined triangles using all points is not a new problem. Almost a century ago, Delaunay laid the foundations for a general solution maximising the minimum angle of all the triangles, which in modern days is referred to as the Delaunay triangulation. Various different implementations of such a triangulation exist in modern days, with one of the algorithms being sweep-hull, having a runtime complexity of $O(nlog(n))$. The novelty is reflected in the radially propagating sweep-hull, based on a radial sort of the points, together with the flipping of triangles to compute the Delaunay triangulation. For further reading, we refer to Sinclair's work describing the full routine. The initial set of points are now transformed into triangles with vertices $[v_1...v_3]$ and surface normal $[n_1...n_3]$, such that,

$$v_{tr} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}, n_{tr} = \begin{bmatrix} n_1 & n_2 & n_3 \end{bmatrix} \qquad \text{Triangle}$$

Figure 4: Triangulating a randomised set of points.

## 3.4 Texture Mapping

Once the triangulation is constructed it needs to be linked to the projected image. This can be achieved by treating the projected image as a texture, and finding colour values for each point in the triangle mesh. The range for texture coordinates is $[0, 1]$, as this is the standard for graphics. The projected image has its width and height pixel coordinates mapped and scaled to fit this range. Furthermore, the point coordinates have been deliberately constructed with this same range, allowing a connection between source and mesh to be set-up without performing additional transformations. To determine the colour of some pixel, we sample from the texture with source coordinates $s$ such that,

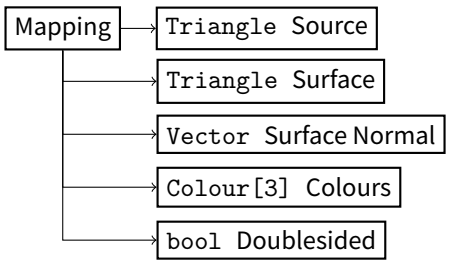
$$c = texture(u = s_x, v = s_y) \qquad \text{Colour}$$

A seemingly simple solution like this does not come without a cost, however, as plain texture mapping often produces images with a diminished quality. This is a clear drawback, as for mirror anamorphosis curvature in mirrors makes this issue even more apparent as it distorts the triangles. Luckily, improving quality in rendering is a classic graphics conundrum, and there exists an abundance of interpolation techniques. For intersections with triangle meshes, given that the colour data of all vertices are known, barycentric coordinates of the intersection point $p$ can be used to interpolate its colour. For some triangle with vertices $[v_1...v_3]$, and edges $[p - v_1...p - v_3]$ subdividing the triangle into areas $[a_1...a_3]$, it holds that,

$$c_p = a_1v_1 + a_2v_2 + a_3v_3 \qquad \text{Barycentric Coordinates}$$

## 3.5 Generating Mappings

In order to re-use any raytracing results, the output first needs to be stored. For now, let us define a basic structure capturing all relevant output that has been produced so far. This structure contains both triangles, the orientation of the triangle on the surface by means of the normal vector, and our set of colours. In addition, a boolean is defined to describe whether a triangle registers intersections on one or both sides. Our mapping contains a list of instances of this new structure, with the our routine so far being,




The conditions for the render are now satisfied. Summarising the process so far; we define points and create triangles from them, and determine the colour for each point. Next, we cast and trace a ray for each point and record the intersections with the surface after passing through the screen and bouncing off the mirror. The complete procedure for computing a mapping from a source image is as follows,

**Initialisation**

1. Initialise output size $(w, h)$, samples per pixel $spp$ and our output files contained in `files`.
2. Set up the camera $c$ with rays originating from $v_{from}$ and being casted to $v_{at}$.
3. Create our scene in the world with objects $(e, m, s)$ for screen, mirror and surface respectively.

**Points and Triangles**

1. Generate a set of points $p_a$ based on the dimensions of the requested output with size $w * h * spp$.
2. Drop points from the set if they do not intersect with objects $(e, m, s)$ to construct $p_t$.
3. Find the Delaunay triangulation $tr$ for the set of points $p_t$.

## 3.6 Image Reconstruction

Texture coordinates are part of a discrete space with range $[0, 1]$, where $(u, v)$ represents horizontal and vertical locations respectively. Our implementation of a texture reading from an image translates pixel locations to texture coordinates, and makes use of the `stb_image` library for parsing the image.[13] The current output to the render provides some arbitrary world space coordinates, and also need to be translated to texture coordinates. We find the minimum and maximum values for the world coordinates of both the projected image and the distorted surface image. We map some coordinate $u$ or $v$ to scale and fit all values to the appropriate range. The function for $v$ is analogous to that of $u$, and the latter is given by,

$$u_{new} = \frac{1}{(u_{max} - u_{min}) * (u_{old} - u_{min})} \qquad \text{Mapping } u$$

Next we iterate over the set of triangles, and determine which of the pixels it overlaps with. This is done by creating a bounding box around the triangle based on its extremes, and testing which points in this bounding box are in the triangle using the 'in-out' test. This test relies on the use of barycentric coordinates $(u, v, w)$, which express a location on a triangle using scalars for each vertex.[12] In short, for every edge we check whether the point is to the left or right, and if all edges match up, the point must be inside the triangle. If the point turns out to be part of the triangle, the respective pixel has its colour updated. If there are multiple colours assigned to a pixel, we increment the value for colour, and later average this by dividing by the total amount of samples. For testing purposes we currently only sample one time when a pixel is part of a triangle, however, by using more sample points the quality can be greatly increased. After computing colour for every pixel the data is converted to a portable pixel map (PPM) image file. This type of format takes an input of

three colours per pixel in the range $[0, 255]$, and we use a utility function to convert and write the result to the file. Taking as input the image size, set of triangles and a location to write output to we implement the remainder of the algorithm as,

**Rendering and Mapping**

1. For each unique vertex in triangle set $tr$ trace a ray such that,
   (a) It propagates from camera to surface with the path being $c \to e \to m \to s$.
   (b) We find the world coordinates of the ray intersecting with surface $s$.
   (c) These coordinates are written to `files`.
2. Assign colours to vertices in $tr$ based on the source texture and readjust sizing if necessary.
3. Rasterise the output image using the coordinates in `files`.
4. For all points we find barycentric coordinates in its respective triangle to determine its colour.
5. Write result to `files` as `ppm` image format.
6. Render the scene using classic raytracing. *(Optional)*

# 4 Results

The algorithm was developed in `C++` and executed on a machine with the a Nvidia GeForce GTX 1070 graphics card, AMD Ryzen 7 3800X processor and 16GB of RAM. Results were written to text files and images in portable pixel maps (PPM) format, containing RGB values for each individual pixel. A `Python` script was used for visualising triangle meshes. The input parameters for the algorithm are `[input_image, world, camera]` with some additional settings `[dimensions, max_depth, samples]` for rendering. Results are composed of components involving either the projected image or the projection on the surface, and reflect various stages of the solution. We provide a visualisation for the set of points we generate, the resulting triangle mesh and a set of morphed images using the mapping derived from this mesh. In addition, we provide a set of classic renders to illustrate various scenes.

## 4.1 Basic Scene

For a frontal view on the mirror our algorithm performs as expected, and matches up with similar existing anamorphic art installations. The scene is set-up with a surface in checkerboard-style, a screen in purple and a mirror cylinder. The screen has full overlap with the mirror from the observer's perspective. After selecting points, the set is trimmed down such that any non-relevant points are omitted when finding the Delaunay triangulation. As reference, the full set of points is later used to generate images of the full scene. For each unique vertex in the triangle we cast a ray into our scene, and record the location of the intersection. This results in a correspondent triangle mesh present on the surface, with an obvious level of distortion visible. By sampling from two source textures, we create transformations of projected image to projection on the surface, and show that they transform distinct textures as equivalent.

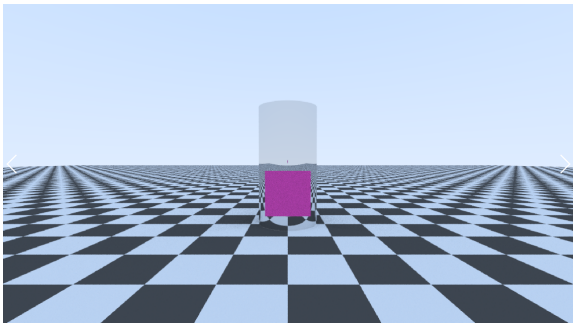
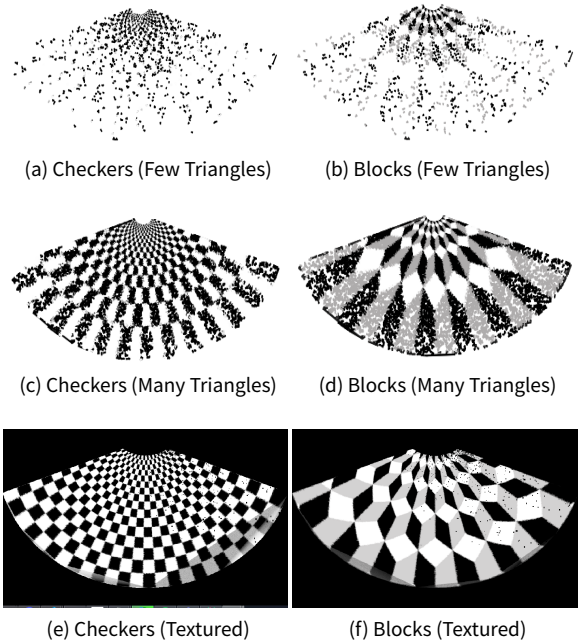


Figure 5: Scene from simple rendering viewpoint.



(a) Checkers (Few Triangles) (b) Blocks (Few Triangles)

(c) Checkers (Many Triangles) (d) Blocks (Many Triangles)

(e) Checkers (Textured) (f) Blocks (Textured)

Figure 6: Applied texture mapping in various stages.

## 4.2 Advanced Viewpoints

For more obscured viewpoints our algorithm performs less optimal, and is usually not desired for reflection art in general. Due to heavy distortion images can stretch out far, and a standard canvas is usually smaller than a standard football field. Distortion can be accounted for with interpolation, but for areas too large this technique takes a performance hit. Certain angles also produce various artifacts such as overlapping colours creating a blur, streaks of black pixels or boundary issues. This is not a scientific limitation and rather one of our own, and we are certain that issues like these can be resolved in due time.

# 5 Discussion

While our approach provides full support for rendering triangle meshes, but presenting mirror objects and surface objects consisting of triangles fell out of scope. This is due to the complexity of constructing such a scene in the first place. There are tools to generate such objects, but this brings difficulty in translating the output from a tool to an input for our own algorithm. Selecting a suitable viewing angle for the observer can be quite problematic, as anamorphosis is inherent to strong viewpoint dependence. Our output image also shows some artifacts, as the implementation for reconstruction is quite low-level and might be susceptible to rounding errors. As reconstruction is currently slow, real-time rendering is not possible. Still, we handle images at a much faster rate than what one would achieve with raytracing the entire scene again, and believe that real-time rendering can be achieved with an optimised implementation of reconstruction.

## 5.1 Responsible Research

The set-up for this research is fully reproducible as there are many different implementations of raytracing. Our method is standardised and does not rely on very specific conditions, and thus similar results are attainable even if the set-up slightly differs. Regardless, the bulk of the techniques used have their traditional, non-anamorphic analogue represented in the academic world already. The same holds for a variety of different artworks, which were used as reference solutions. As for the transparency of the results, the figures are a one-to-one visualisation of the underlying numbers generated by our algorithm. The development of the technique was conducted in an ethical and fair manner, and we hope to broaden the perspective on anamorphosis of prospective artists and researchers alike.

# 6 Conclusion

We introduced a solution to generating mappings from projected image to surface in mirror anamorphosis. Our approach implements and algorithm which starts by generating a set of points based on the desired quality. It triangulates them to a two-dimensional triangle mesh using Delaunay triangulation. Using raytracing, the mesh is projected onto the mirror through a screen, bounding the proportions of the output. Any ray intersecting with the surface after being reflected off the mirror object is recorded in a mapping with its respective texture coordinates. Based on this mesh, we do not have to execute our raytracing algorithm again if the projected image is changed. We use texture mapping to transform any projected image into the distorted projection on the surface and achieve speeds much faster than initial raytracing. This makes on the fly changes to new artworks possible, and provides a baseline for rendering real-time anamorphoses in industries such as film and video gaming. Future work would involve optimising the current implementation for faster triangle intersection, texture mapping and point selection. Lastly, we want to test mesh-division[14], point sampling[15] and nested level of detail[16] as adaptive strategies for anamorphic raytracing.

# References


[1] F. De Comité, "A general procedure for the construction of mirror anamorphoses," in *Bridges 2010*, Jul. 2010.

[2] J. Hunt, B. Nickel, and C. Gigault, "Anamorphic images," *American Journal of Physics - AMER J PHYS*, vol. 68, Mar. 2000. DOI: 10.1119/1.19406.

[3] C. Gabriel-Randour and J. Drabbe. (2001). Cabri et les anamorphoses, [Online]. Available: http://www.uni-klu.ac.at/stochastik.schule/ICTMT_5/ICTMT_5_CD/Special%20groups/Gabriel_frz.htm (visited on 06/25/2021).

[4] P. Shirley, *Ray tracing in one weekend*, Dec. 2020. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[5] N. Farenc, M. Roelens, and C. Fauriel, "Creating special effects by ray-tracing with non classical perspectives," Oct. 2000.

[6] K. Hormann, "A quadrilateral rendering primitive," Jan. 2004, pp. 7–14. DOI: 10.1145/1058129.1058131.

[7] B. Delanue, "Sur la sphere vide. a la memoire de georges voronoi," *News of the Academy of Sciences*, no. 6, pp. 793–800, 1934.

[8] J. Sjoholm. (Jun. 2020). Creating optimal meshes for ray tracing, [Online]. Available: https://developer.nvidia.com/blog/creating-optimal-meshes-for-ray-tracing/ (visited on 06/26/2021).

[9] D. Sinclair, "S-hull: A fast radial sweep-hull routine for delaunay triangulation," Mar. 2016.

[10] P. S. Heckbert, "Survey of texture mapping," *IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 56–67, 1986. DOI: 10.1109/MCG.1986.276672.

[11] J. Amanatides and K. Choi, "Ray tracing triangular meshes," in *Proceedings of the Eighth Western Computer Graphics Symposium*, vol. 43, 1997.

[12] Scratchapixel, *Ray tracing: Rendering a triangle*, 2016. [Online]. Available: https://www.scratchapixel.com/.

[13] S. T. Barrett, *Stb*, https://github.com/nothings/stb, 2021.

[14] Y.-M. Ji, H. Yeom, and J.-H. Park, "Efficient texture mapping by adaptive mesh division in mesh-based computer generated hologram," *Opt. Express*, vol. 24, no. 24, pp. 28 154–28 169, Nov. 2016. DOI: 10.1364/OE.24.028154.

[15] G. Schaufler and H. W. Jensen, “Ray tracing point sampled geometry,” in *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Berlin, Heidelberg: Springer-Verlag, 2000, pp. 319–328, ISBN: 3211835350.

[16] J. Xia, J. El-Sana, and A. Varshney, “Adaptive real-time level-of-detail-based rendering for polygonal models,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 3, pp. 171–183, May 1997. DOI: 10.1109/2945.597799.