Constructing 3D Perspective Anamorphosis via Surface Projection

Tiffany C. Inglis

D2L, Waterloo, ON, Canada; piffany@gmail.com

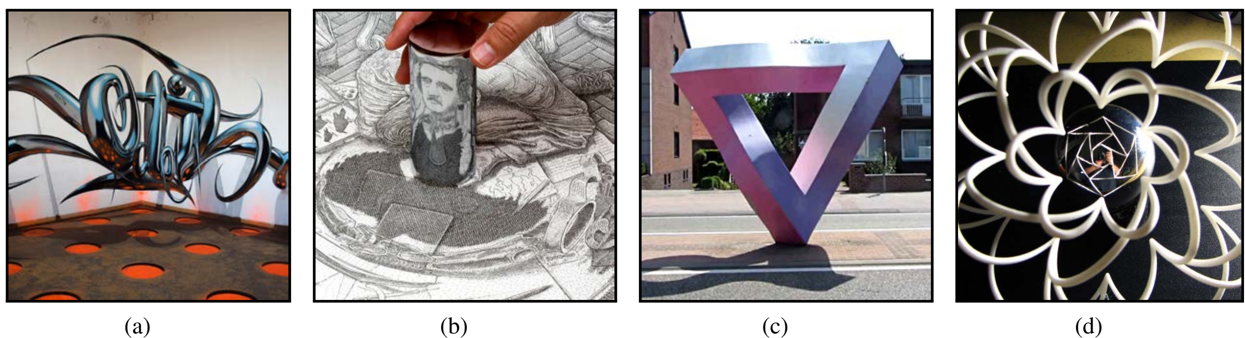
Abstract

3D perspective anamorphoses, as defined in this paper, are 3D objects that create anamorphic illusions when viewed from a particular perspective. We present an algorithm of constructing 3D perspective anamorphoses by projecting onto surfaces.

Introduction

An *anamorphosis* is an object that looks distorted unless viewed in a special way. The method of distortion characterizes two main categories of anamorphic illusions: *perspective* (viewed from a particular perspective) and *mirror* (viewed through a mirror). The object of distortion also falls into two types: 2D images and 3D models. Figure 1 shows an example from each category. Odeith's [3] and Orosz's [4] work are both 2D anamorphoses since they are 2D drawings that "come to life" (either by appearing 3D or revealing hidden images) when viewed from a certain angle or through a mirror. 3D anamorphoses include Hamaekers's sculpture [5] depicting an impossible triangle with curved edges, and De Comité's sculpture [1] also using curved edges to create the illusion of a polygonal structure in the spherical mirror.

A 2D anamorphosis is essentially an image projected onto some surface either via perspective projection or mirror reflection. For simple surfaces and mirrors, the design can be drawn manually using distorted grids. With more complex inputs, dedicated software such as *Anamorph Me!* [2] can automatically apply the necessary distortions. An artist may also use a projector in combination with projection mapping software to directly project their designs onto surfaces with complex geometry.



(a) (b) (c) (d)

Figure 1: (a) *Anamorphic Chrome Letters Ground Reflection* (2015) by Sérgio Odeith, (b) *The Raven* (2006) by István Orosz, (c) *Unity* (1995) by Mathieu Hamaekers, and (d) *Hexagonal Flower* (2013) by Francesco De Comité.

In contrast, designing 3D anamorphoses is a much more open-ended problem since the object of distortion is no longer a flat image. A sculpture of an impossible triangle, for instance, can take on many forms and still produce the same illusion when viewed from the intended perspective. De Comité's algorithm [1] succeeds in creating 3D mirror anamorphoses and, in this paper, we will be focusing on developing a tool

to help visualize and construct 3D perspective anamorphoses. Since the design space for 3D anamorphoses is enormous, we will artificially restrict the design to something that can be projected onto a surface. As such, our main goal is to create a tool that projects 3D objects onto a 3D surface, satisfying the following conditions:

- Simplicity: Simple design creation, viewing from different perspectives, and component manipulation.
- Convenience: No additional hardware such as projectors is needed.
- Compatibility: 3D objects can be imported and exported from 3D modeling software.
- Realization: Supports 3D printing and paper craft construction based on the user's designs.

Software Design

The source code and full documentation of the software is available online. Here we provide a brief description of the main features.

Scene construction. Each design is based on a scene that consists of 3D objects projecting onto a surface from an observer's view. The design is anamorphic in nature because it will look like the set of 3D objects when viewed from the observer, and distorted otherwise. The 3D objects and the surface contain edges and faces specified as Wavefront OBJ files. In addition, the projected objects may have coloured faces (by extending the OBJ file format) so that the resulting projection can contain colours. For human readability, we save the scene as a JSON file.

3D mode. Our software provides four modes to view and interact with the design. The 3D mode allows the user to graphically manipulate the scene objects together or separately, and re-project from different observer positions. This mode is useful for seeing your design from various perspectives and fine-tuning it until the desire scene and view parameters are achieved.

Continuous projection mode. Imagine the objects are no longer projected from the observer, but rather from a light source. In this mode, it is as if we are looking the shadows casted by a moving light source. It is a useful way to explore the space of possible projections given the scene layout.

Paper craft mode. The mode displays an unfolding of the surface containing the projections, so that the design can be cut out and assembled to create a paper sculpture showcasing the illusion. Since creating an unfolding from an arbitrary polyhedron is a difficult problem, our program requires the unfolding structure (given as a tree of connected faces) to be manually specified. For convenience, we supply unfoldings for primitives such as spheres and cubes.

3D printing mode. To create a 3D perspective anamorphic sculpture, we can 3D print the projected shapes that lie on the projection surface (but without the actual surface). Although the projections may contain both edges and faces, we typically omit the faces because 3D printed surfaces usually have low structural integrity. If necessary, they can be printed as a wireframe mesh. Converting our design to a 3D printable object is not natively supported by our program, but rather through a Mathematica script that we provide. Basically, the script extracts the edges we want to print, thickens them as tubes, and exports them in STL format.

The Mathematics

After describing the algorithm behind our program on a high level, we now discuss some of the highlights in detail. Figure 2 outlines the main algorithm. We first define the scene (see Figure 3a) with a surface and some objects to project onto it. The surface contains only triangular faces (if not, we triangulate them),

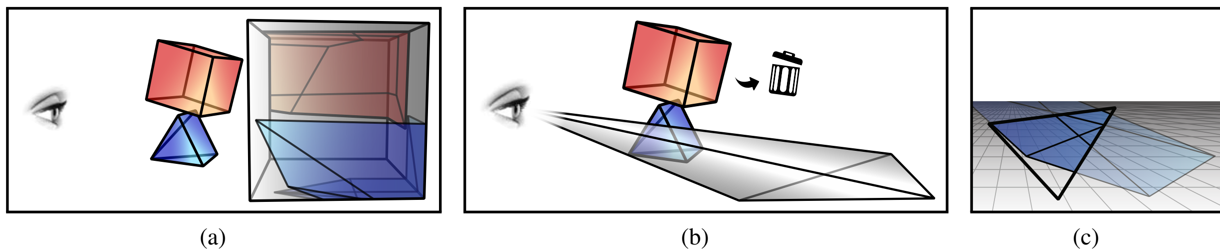
while the objects can contain both edges and faces.[1] Then for each triangle on the surface, we compute the associated projection from the objects. The triangle projection is split into three steps: (1) culling of the view frustum, (2) projecting onto the triangle plane, and (3) clipping to the triangle. In subsequent discussions, I will refer to the edges and faces of an object as its *elements*.

```
main()

defineScene(surface, objects)
triangles = surface.triangulate()
elements = objects.getEdgesAndFaces()
projection = []
forEach(triangle in triangles):
  culled = viewFrustumCulling(elements, triangle)
  projected = projectOntoTrianglePlane(culled, triangle)
  clipped = clipToTriangle(projected, triangle)
  projection.add(clipped)
display(surface, objects, projection)
```

Figure 2: Outline of the main algorithm.

View frustum culling. When projecting an object onto a triangle, only a small subset of its elements may fall inside the triangle. Elements outside the view frustum—the "cone of vision"—can be culled to reduce the amount of processing required in later steps. Figure 3b shows an example where the cube elements are discarded for being outside the frustum while the pyramid elements are retained.



(a) (b) (c)

Figure 3: (a) The scene contains two objects projected onto a surface from the observer. (b) For each triangle on the surface, the object elements outside the view frustum are discarded. (c) The remaining elements are then projected onto the triangle plane, and clipped to the triangle.

Projecting onto triangle plane. After culling, we want to project the remaining elements onto the target triangle. This is done by first projecting onto the plane containing the triangle, and then clipping to the triangle's boundary (see Figure 3c). How do we project a polygon onto a plane? With a single point, it is simple; just cast a ray from the observer through the point and computes its intersection with the plane. By extrapolation, we could project each vertex on a polygon and then discard those that do not intersect with the plane. But does that really work?

Consider this scenario: suppose we wish to project a tall triangle from the observer onto a horizontal plane (see Figure 4a). Visually, we can see that its projection is a quadrilateral or, more precisely, the portion of the projected triangle that sits below the horizon. But we had projected each vertex separately and

[1] Each face must be a simple polygon.

discarded the top vertex for being above the horizon, we would be left with only a line segment that joins the bottom two vertices, which is incorrect.

The problem is that perspective projection of a polygon cannot be handled on a per-vertex basis because the observer-vertex rays are all pointing in different directions, so it is possible that only a subset of the polygon's vertices project onto the plane. In contrast, under parallel projection, all the rays would be parallel. Hence a polygon would either be entirely projected or not projected at all.

One solution is to somehow factor in the horizon when computing the projection, but that involves the hassle of dealing with points at infinity. It also seems unnecessary considering the subsequent clipping step will remove anything above the horizon anyway. We use an easier approach that keeps track of the projected triangle consisting of two real vertices on the plane and one "fake" vertex above the horizon.

Observe that these fake projection points actually have physical locations behind the observer. Figure 4b shows a point that projects forward onto what we consider a real projection, and a point that projects backward onto a fake projection. We can visualize the forward and backward projections together by joining the two planes at the horizon, as shown in Figure 4c. In this setup, the bottom plane is the forward projection plane and the top plane is the backward projection plane reflected horizontally and vertically. Now the triangle from Figure 4a can be projected onto both planes, as shown Figure 4c.

Notice there is something peculiar about how the line segment AB is drawn. Since A and B are respectively in front of and behind the observer, we expect the line segment AB to wrap around the viewport. But instead A and B are joined at the horizon, as if the line segment has been inverted. This phenomenon happens whenever we have a line segment joining a forward projected point to a backward projected point. We need to carry forward the information so the clipping algorithm can properly handle these edges.

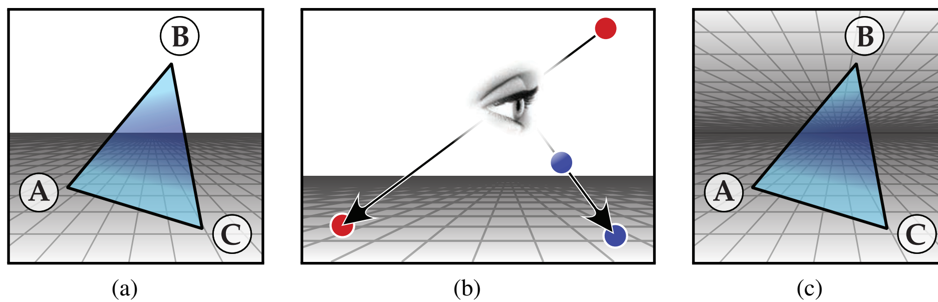


Figure 4: (a) A triangle may not completely project onto a plane. (b) By considering both forward and backward projections, (c) the flagged projection can be described as a projection spanning both planes.

Currently, each projected point is represented by a 3D vector (x, y, z). To keep track of the projection direction, we extend it to a 4D vector (x, y, z, f) with *projection flag* f, where $f = 1$ denotes a forward projection and $f = -1$ a backward projection. We shall refer to this as an *flagged projection*. With the projection flag, we can now project a line segment or a polygon onto a plane vertex-by-vertex.

Clipping flagged projections. Next, we want to clip the flagged projection to a triangle. Let's first simplify the problem by performing a change of basis so that the triangle becomes the unit triangle (with vertices $(0, 0)$, $(1, 0)$ and $(0, 1)$) on a 2D plane. Consequently, each flagged projection vertex become a 3D vector of the form (x, y, f) where (x, y) is the transformed position and f is the same projection flag pre-transformation.

For a regular unflagged projection, we can use the Sutherland-Hodgman algorithm [6], which clips a simple polygon to a convex polygon (a triangle in our case). It can of course clip a line segment as well since a line segment is just a polygon with two vertices. For a flagged projection, however, some preprocessing may be required.

Suppose there is a flagged line segment that we wish to clip to a triangle in the forward projection plane. If a line segment contains two forward projected points (i.e. both with $f = 1$), then we simply apply regular clipping. If both points are backward projected (i.e. both have $f = -1$), then the line segment is not in the forward projection plane at all, so we discard it. The more interesting case is when one endpoint has $f = 1$ while the other has $f = -1$. We saw earlier that such a line segment extends to infinity in both projection planes. In other words, the line segment must be *inverted* as illustrated in Figure 5b before we can clip it. The inversion of line segment FB is basically the result of subtracting line segment FB from line FB.

Since inversion is only applied to a line segment containing a forward projected point and a backward projected point, the process always results in a forward ray and a backward ray. As the backward ray lies entirely in the backward projection plane, it too can be discarded prior to clipping. Thus, in practice, we only keep the forward ray after applying inversion (see Figure 5a). Clipping flagged polygons requires similar preprocessing. We first convert the polygon into an ordered list of edges, apply inversion to the appropriate edges, and then recombine the processed edges.

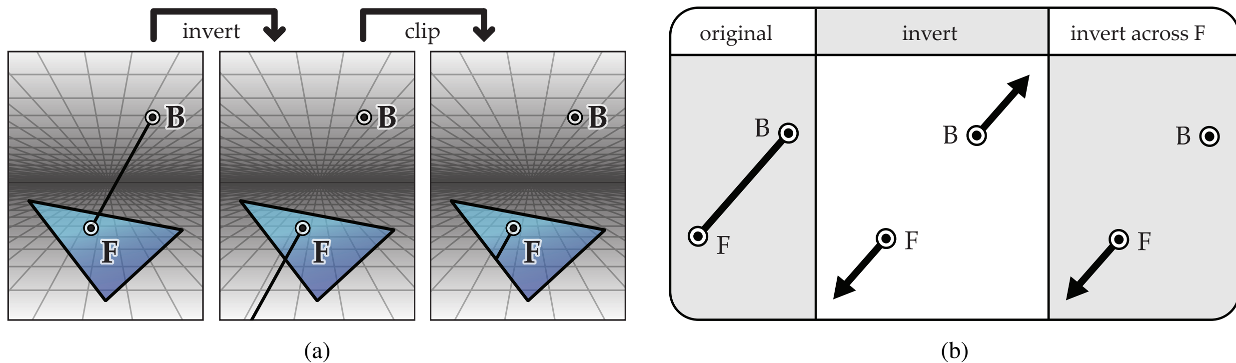


Figure 5: (a) Clipping a flagged element may require inverting before clipping. (b) For the purpose of clipping, inverting across the forward projected point F is sufficient.

To summarize, clipping flagged elements involves inverting some elements and then clipping via the Sutherland-Hodgman algorithm. The steps are outlined in Figures 6 and 7. Note that "inverting edge FB across F" means inverting line segment FB and only keep the ray containing point F while discarding the ray containing point B, as illustrated in Figure 5b.

```
clipEdge(edge, triangle)

[p1,p2] = edge.getVertices()
[f1,f2] = edge.getFlags()
if(f1<0 && f2<0) return null
if(f1>0 && f2<0) edge.invertAcross(p1)
if(f1<0 && f2>0) edge.invertAcross(p2)
return sutherlandHodgman(edge, triangle)
```

Figure 6: How to clip a flagged edge.

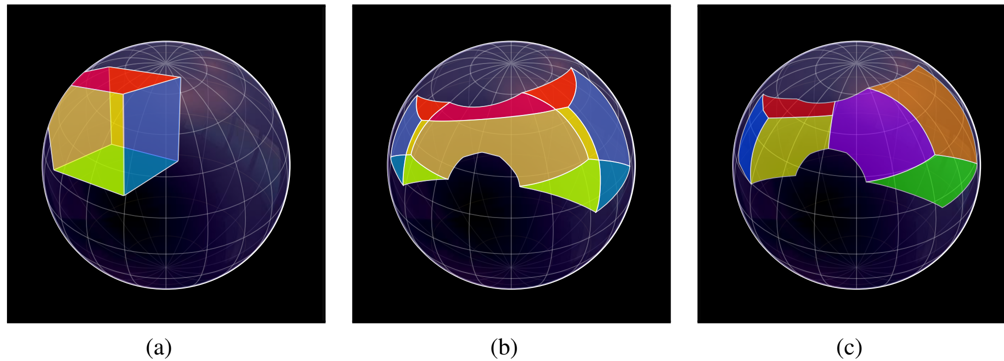
```
clipPolygon(polygon, triangle)

edges = polygon.getEdges()
newEdges = []
forEach(edge in edges):
  newEdge = clipEdge(edge, triangle)
  newEdges.add(newEdge)
return new Polygon(newEdges)
```

Figure 7: How to clip a flagged polygon.

Projection targeting. With how the algorithm is currently structured, the projection can end up looking a little flat. For example, if we projected a cube onto a sphere (see Figure 8a), the resulting projection is nothing more than a distorted image of the cube (see Figure 8b). Even though we started with a 3D object, the projection flattens it onto the surface. The fact that we could have gotten the same result using just an

image of the cube is a somewhat unsatisfying.



(a) (b) (c)

Figure 8: (a) A cube projected onto a sphere with projection targeting (b) disabled, and (c) enabled.

One way to make the projection less flat is to give it front and back faces, just like the object from which it originated. Consider the cube example again. By computing the outward normals, we can split its faces into front faces and back faces. The same can be done with the spherical surface. Now we target the projections so that front faces only project onto front faces and back faces only project onto back faces. The result is a projection structured more similarly to the original cube (see Figure 8c).

Projecting edges this way is more difficult because, unlike faces, they do not have well-defined outward normals. For isolated edges, we simply project them onto all surfaces. However, in a polyhedron, each edge has a direction associated with its two neighbouring faces. Thus we can target their projections as follows:

1. A front edge (i.e. between two front faces) should project only onto front faces.
2. A back edge (i.e. between two back faces) should project only onto back faces.
3. A side edge (i.e. between one front face and one back face) should project only onto all faces.

This front-to-front and back-to-back scheme is but one of many ways to target projections. In the next section, we will show an interesting example that uses a front-and-back and back-to-front scheme.

The Art

In Figure 9, we show some artwork created with help of the software. For each design, 3D models are first created as inputs to the program. After adjusting some parameters, they are either exported as raster images, or rendered the 3D print models from different perspectives. The raster images are redrawn by hand as vector graphics with various effects added.

Figures 9a, 9b, 9c show a sphere projected onto a cube, a cube projected onto a sphere, and an impossible triangle projected onto a torus. In all three examples, the front-to-front and back-to-back projection targeting scheme is applied. For the impossible triangle example, the object projected actually contains two layers—an impossible triangle and its enantiomer—such that the resulting front and back projections are two impossible triangles seamlessly joined together.

In Figure 9d, a cube is projected onto a helix. Since the helix's normal vectors are all pointing in the upward towards the observer (i.e. all surface faces are front facing), we disabled projection targeting so that the cube would project onto all the layers. Notice that the upper projection faces are smaller than the lower ones; this is to compensate for the foreshortening in perspective projection. When an object projects onto

a large portion of the surface, the resulting design shows more of the surface structure. For this reason, we chose to project a snowflake[2] onto a cube (see Figure 9e) since its diagonal cross section is a hexagon.

Now we will show some designs as 3D models. First we have a dodecahedron projected onto a bumpy sphere (see Figure 9f). The bumpiness manifests as curly edges as we rotate the model. Next, we wanted to design an impossible figure with some unexpected connectivity properties. We chose the impossible cube, which contains two front edges behind two back edges. To achieve this effect in our program, we used a front-to-back and back-to-front projection target, which brings the back edges to the front. Figure 9g shows the final result as the model rotates.

Future Work

Using projection as a means of creating 3D perspective anamorphoses opens up many possibilities, but also introduces inherent limitations. When projecting, say, a cube on a sphere, the cube needs to be large enough so that the front and back projections connect, but not so large that it would be significantly truncated. Figure 9a shows a cube meticulously tweaked so that it is approximately inscribed in the sphere. Resolving this issue of preserving connectivity while avoiding truncation would greatly improve the usability of our software.

Projection targeting can be further improved upon. Targeting based on normal vectors is too restrictive but, on the other hand, manually specifying targets for each element would be too tedious. A good compromise would involve some convenient way of programmatically specifying projection targets to suit different scenarios. Our longterm goal is to use the projection framework as an initial step in the design process for 3D perspective anamorphosis. We want to create tools to further edit and refine the designs without destroying the intended illusion. Such tools would allow us to create designs with multiple intended views.

Acknowledgements

I would like to thank David Swart for the consultation on projection mapping and related research, and Prof. Craig Kaplan for advice on 3D printing.

References

[1] F. De Comité. "A New Kind of Three-Dimensional Anamorphosis." *Bridges Conference Proceedings*, Coimbra, Portugal, July 27–31, 2011, pp. 33–38. http://archive.bridgesmathart.org/2011/bridges2011-33.html.

[2] P. Kent. *Art of Anamorphosis Software*. 2001. https://www.anamorphosis.com/software.html.

[3] S. Odeith. *Anamorphic Chrome Letters Ground Reflection – 2015 Odeith*. 2015. http://www.odeith.com/anamorphic-chrome-letters-ground-reflection-2015-odeith/.

[4] I. Orosz. *RAVEN-MORPH*. 2015. http://utisz.blogspot.ca/2015/10/raven-morph.html.

[5] A. Seckel. *Masters of Deception: Escher, Dalí & the Artists of Optical Illusion*, Sterling Publishing Company, Inc., 2004, pp. 131–135.

[6] I. Sutherland and G. Hodgman. "Reentrant Polygon Clipping." *Communications of the ACM*, 1974, vol. 17, pp. 32–42.

[2] Snowflake design obtained from http://demonstrations.wolfram.com/FractalCurves/.

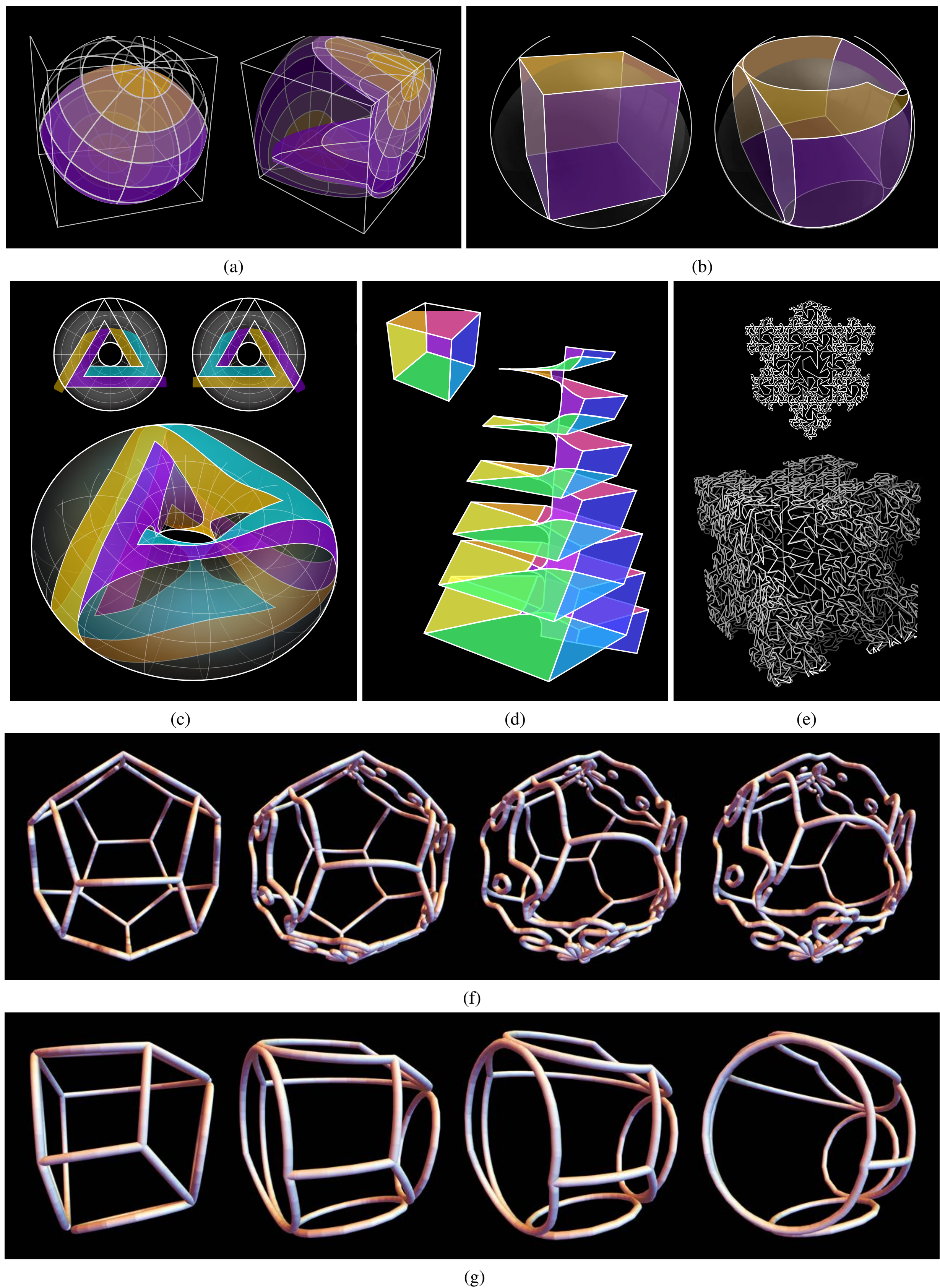


Figure 9: (a) A cubic sphere, (b) a spherical cube, (c) a toroidal impossible triangle, (d) a helical cube, (e) a cubic snowflake, (f) a dodecahedron on a bumpy sphere, and (g) an impossible cube on a sphere.