Joint Virtual Reality Conference of EuroVR - EGVE (2011)
S. Coquillart, A. Steed, and G. Welch (Editors)

Short Paper: View Dependent Rendering to Simple Parametric Display Surfaces

Pawan Harish[†] and P.J. Narayanan[‡]

Center for Visual Information Technology, International Institute of Information Technology - Hyderabad, India

Abstract

Computer displays have remained flat and rectangular for the most part. In this paper, we explore parametric display surfaces, which are of arbitrary shape, but with a mapping to a 2D domain for each pixel. The display could have arbitrary curved shapes given by implicit or parametric equations. We present a fast and efficient method to render 3D scenes onto such a display in a perspectively correct manner. Our method tessellates the scene based on the geodesic edge length and a user-defined error threshold. We also modify scene vertices, based on per-vertex ray casting, so that the final image appears correct to a user's viewpoint. The ray-surface intersection procedure, geodesic length computation and 2D image mapping are assumed to be known for the given surface. We exploit the tessellation hardware of the SM 5.0 GPUs to perform the error checking, polygon splitting, and rendering in a single pass. This brings the performance of our approach closer to rasterization schemes, without needing ray tracing. Our scheme does not interpolate pixels, ensuring high quality. We demonstrate real display prototypes and show scalability of our system using simulated scenarios.

1. Introduction

Displays have changed much over the years with advances in the basic technology, color gamut, vertical refresh rate, power consumption, pixel resolution, etc. They are still mostly flat, inactive, and rectangular windows meant to be interacted with by sitting in front. Three dimensional displays and touch screens have enabled new interaction methods in recent times. In this work, we consider displays of arbitrary shape. As computer generated imagery becomes more prevalent, different surfaces in one's environment may be turned into a display surface using projection or by fixing suitable display material on them. This may include parts of one's office or home, lobbies and other public spaces, as well as curiosity surfaces in museums and other places.

A surface of arbitrary shape can be covered with display elements or pixels with high density. These can be thought of as being fixed on the surface. We define *parametric display surfaces* as those in which the pixels map to a suitable two-dimensional domain. This provides each pixel with a unique id or index. The display and an image in its target domain are equivalent and we can drive the display by generating the image, called its *base texture*. A simple rectangular domain or a subset of it is most convenient when using modern graphics pipelines.

We present a view dependent scheme to render 3D scenes to general parametric display surfaces. The display surface is specified as a set of equations that define its shape implicitly or parametrically. We assume the mapping of actual display pixels from its base rectangular texture is already defined. We, therefore, focus only on rendering to the base texture correctly from the point of view of a user whose head is tracked. Such a display can be thought of as a generalization of fish tank virtual reality (FTVR) display to an arbitrary shape. We demonstrate our rendering scheme using a few real curved surfaces of algebraic form, namely, sphere and cylinder. We also show more complex algebraic display surfaces using simulations. We achieve over 100 FPS on a scene with close to a million triangles when rendered to a spherical parametric display.

2. Related Work

Curved surfaces have been used to display 2D images using texture mapping [BWB08, RWF98]. Extending this

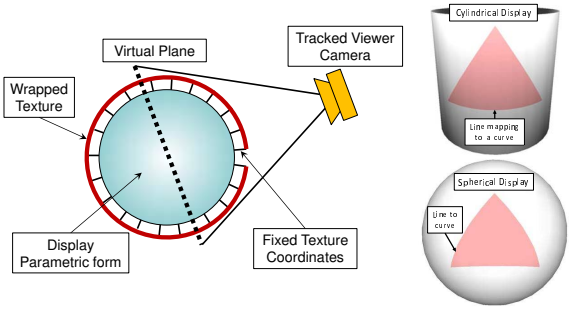
[†] harishpk@research.iiit.ac.in
[‡] pjn@iiit.ac.in

DOI: 10.2312/EGVE/JVRC11/027-030

approach to display 3D scenes is not trivial. Bimber et al. [BWEN05] show stereoscopic projection on any surface using the two pass texture mapping approach. Fish tank virtual reality allows realistic exploration of 3D scenes using head tracking. Each planar facet of the display shows an image based on the viewer location such that the combined effect produces the illusion of virtual objects placed in viewer space or vice-versa. Though the notion of FTVR displays is old [Fis82], one of the first implementations was the CAVE virtual environment [CNSD93]. Five back-lit planes were projected on to create a virtual enclosure. Off axis rendering [Dee92] was used to generate projection matrices through which the scene was rendered independently for each plane. FTVR displays have followed this template with alterations to the basic framework. Cubby [DOS01, FDO02] added manipulation tools to interact with the display. Cubee [SVF06] created an inside-to-outside viewing cube for physics based visualizations. PCubee [SLF10] hand held display enabled bimodal interaction with a personal display. Iwata [Iwa04] created a rhombic dodecahedron FTVR display to facilitate better pixel and space efficiency. In our earlier work [HN09] we give a GPU algorithm for better image generation for multi-planar display surfaces that can approximate arbitrary surfaces. The present work focuses on rendering to a display that is exactly defined parametrically.

3. Rendering on to Parametric Display Surfaces

Rendering to a parametric display surface is equivalent to rendering to its base texture, which is wrapped around the surface of the display (Figure 1(a)). This texture is the target for our rendering pipeline. Given a curved display shape, it can be seen that lines on the texture can become curves (Figure 1(b)). The position of end vertices may also change due to the mapping process. Ray tracing the scene to visible pixels is the natural way to render to parametric display surfaces. This produces correct results at heavy costs. Our rendering scheme approximates this using rasterization and tessellation to a user-specifiable error bound. We modify the scene to be displayed so that a perspectively correct view is seen from the observer's viewpoint. Two attributes must be preserved to do this on an arbitrary surface: Linearity and vertex location in the viewer's eye coordinates must be preserved.



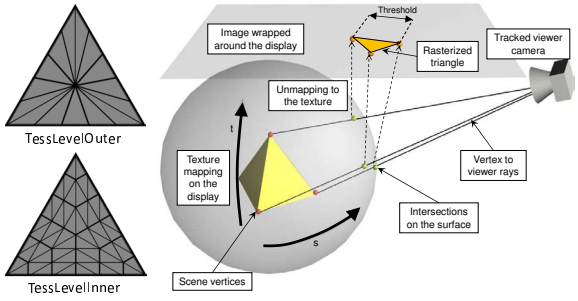
(a) Display parametric form, texture image and viewer camera in the same coordinate frame

(b) Lines mapping to curves

Figure 1: *Parametric display form along with texture wrapped around the display, note line on the texture mapping to a curve on the display surface.*

3.1. Triangle Division to Preserve Linearity

In order to preserve linearity on the surface, lines in the scene should be pre-warped in the texture image, to compensate for the local surface curvature. Correct solution to this problem requires non-linear rasterization of the scene to the texture. It can be observed that the curvature effects are negligible for small line segments in the world. In the limit case, ray tracing to base texture pixels will provide correct results. We approximate non-linear rasterization by subdividing or tessellating large triangles on the fly. This breaks line segments into smaller ones. We exploit the tesselation unit available in the Shader Model 5.0 GPUs to do this. The subdivision and rendering are preformed in a single pass. The tessellation control shader is used to divide a given triangle into smaller triangles. The tessellation hardware divides edges of a given triangle using *TessLevelOuter* parameter. For a given triangle three TessLevelOuter values are passed to the tessellation hardware, one for each edge. The tessellator divides each edge into equal size segments using these values and forms smaller triangles as shown in Figure 2(a), top. The length of a line segment to be drawn is fixed based on the local display geometry. To do this, we intersect the ray from the viewpoint to each triangle vertex with the exact display surface. We assume a ray intersection solution is available for the given shape. Ray tracing methods using shaders are now available for higher order surfaces [SN10]. The geodesic length of the triangle sides determine the level of subdivision so that a user-specified error is not exceeded. Geodesic length computation depends on the surface used and may differ for various shapes. We use the great circle distance for our spherical prototype. The tesselation parameters are computed adaptively



(a) Tessellation Levels

(b) Finding intersection on the surface along with the corresponding pixel value

Figure 2: *Tessellation levels, and finding edge length and vertex location based on per vertex ray casting.*

based on the triangle edge lengths and the error threshold. Figure 2(a)(top) shows that even after dividing the edges into smaller segments the newly formed triangles may end up with edges greater than the threshold. The hardware tessellator uses the *TessLevelInner* parameter to divide the inside of a given triangle. A single value is passed to set this. We set TessLevelInner as the maximum of TessLevelOuter values to ensure that all triangles will have edge lengths less than or equal to the threshold, as shown in Figure 2(a)(bottom). The triangle tessellation control shader is shown in Algorithm 1.

Algorithm 1 Tessellation Control Shader

1: Input: Vertices[3], Threshold
2: Compute geodesic length of each edge on the surface as D[1 to 3] based on ray-casting.
3: TessLevelOuter[1 to 3] = D[1 to 3]/Threshold
4: TessLevelInner = max(TessLevelOuter[1 to 3])
5: Output: TessLevelOuter[3], TessLevelInner

3.2. Modifying Scene Vertices

Subdivision of the triangles is not enough to ensure correct rendering. The resulting vertices should be moved to ensure an undistorted image. Modified positions depends on the local display surface and the viewer location. We compute new vertex positions using per vertex ray-casting, finding intersection of each eye-vertex ray with the display surface and changing the vertex coordinates to the new location. The vertices can then be mapped to the texture space using the mapping from display surface to the base texture for rasterization, as shown in Figure 2(b). We move the newly formed vertices in the normalized canonical space. The tessellation evaluation shader is used to generate vertex coordinates in this space. Each newly formed vertex finds a ray to the viewer location. The ray is intersected with the surface and the intersection point is converted to a (s,t) coordinate in $[0,1]$ range using the base texture mapping. The vertex then sets its (x,y) coordinate to the new (s,t) location while the depth value is computed using the viewer camera. The tessellation evaluation shader is given in Algorithm 2.

Algorithm 2 Tessellation Evaluation Shader

1: Input: Vertex, Viewer location, Viewer camera
2: Find the ray from viewer head position to the vertex
3: Intersect ray with parametric equation of the surface
4: Convert intersection point (x,y,z) to (s,t) in $[0,1]$ range using texture mapping
5: Set vertex x-coordinate as s and y-coordinate as t
6: Compute vertex z-coordinate based on viewer camera
7: Output: Vertex

3.3. The Overall Rendering Pipeline

The overall rendering pipeline (Figure 3) works in a single pass to render a given scene onto the parametric display. The pipeline maps to the SM 5.0 pipeline by replacing tessellation control shader and tessellation evaluation shaders for the oval boxes. A head tracker sends the viewer location to the system. The surface shape and the mapping to the base texture are known to the shaders. Triangles are subdivided and their vertices are modified by the shaders. The resulting mesh is then sent down to the rasterizer to generate the base texture image which is mapped on to the display surface using fixed texture coordinates. The image when wrapped around the display appears perspectively correct from the viewer's point of view. Figure 4 shows the 3D scene, the rendered base texture image with and without tessellation and the view on the surface as observed from the viewer's point of view.

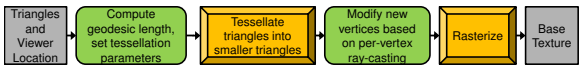


Figure 3: *Rendering pipeline for view dependent parametric display surfaces.*

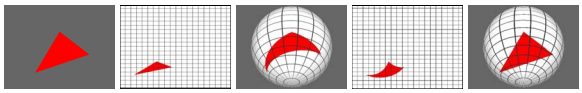
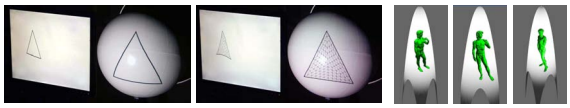


Figure 4: *Tessellation and ray casting to generate inverse curve on the texture. Figure (a) the scene, (b) texture without tessellation, (c) non-tessellated texture on display, (d) texture using tessellation and (e) tessellated texture on display.*

4. Display Prototypes

We demonstrate our rendering mechanism using projected and simulated setups. Projected setups allow only a limited number of pixels on the surface, but still demonstrate the entire process. Our prototypes show $30-40\%$ of the projected pixels on the surface. The base texture is wrapped around a virtual replica of the physical display and is projected on to the calibrated physical display using a calibrated projector. For the simulated displays, we use a base texture resolution of 64M pixels to support many types and sizes of displays. Head is tracked using two infrared head trackers, TrackIR5, with a refresh rate of 120Hz the latency is minimized. Figure 5 shows various scenes in spherical and cylindrical prototypes. A single GTX470 GPU is used to tessellate and render the scene. In Figure 6(a) we show the effect of tessellation on a real display. Lines map to curves when projected on to a spherical display without tessellation, even when vertex positions are corrected based on viewer location. Edges of the triangles become straight with tessellation

Figure 5: *Projection based display prototypes.*



(a) Rendering with and without tessellation. LCD shows the texture image.

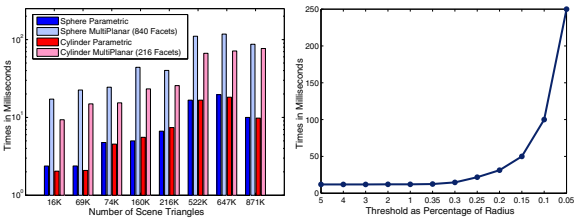
(b) Simulated Display

Figure 6: *Effect of tessellation on a real display. A simulated display with surface equation $y = x^2 + z^2$.*

enabled along with vertex position correction from the point of view of the observer. The display shown in Figure 6(b) has parametric form $y = x^2 + z^2$, and a high curvature. Our rendering pipeline is capable of handling surfaces with high curvature. The curvature of the surface is compensated for by using a smaller distance threshold for tessellation (1.2% of the height of the display). Decreasing the threshold to small percentages has minimal affect on performance as reported in Section 5.

5. Performance Evaluation

We demonstrate the scalability and performance comparison of our rendering pipeline in this section. We use a single Nvidia GTX 470 GPU with 1.2GB of RAM on an Intel Q6600 processor with 2GB RAM as our testbed for all the experiments reported in this section. Figure 7(a) compares the rendering times of our method with our previous approach, approximating the surface using multiple planes [HN09], for varying scene complexity. Typically we see an increase in rendering times with increasing number of scene triangles. However, the reverse may also happen: the scene with 871K triangles shows faster rendering as compared to scenes with lesser number of triangles. This happens because a scene with a large number of triangles will have smaller triangles, such lie within the threshold range and are thus not tessellated. It can also be seen our method is orders faster than [HN09] because of single pass rendering. We see an average speed gain of 20× over our previous method. Figure 7(b) explores the effect of varying tessellation threshold for the 817K triangle scene on the spherical display. It can be seen that even at distances as low as 1% of the radius of the sphere, the performance penality due to



(a) Rendering times with varying scene complexity compared to [HN09]. Threshold = 0.01×radius.

(b) Effect of reducing the threshold. Display: Sphere, Model: Dragon (871K triangles).

Figure 7: *Performance analysis of our rendering pipeline*

tessellation is negligible. Performance degrades after 0.25%, which is a very small distance as compared to the size of the display. Visually no variation beyond 1% of the radius as tessellation threshold was observed for any scene.

6. Conclusion and Future Work

We presented an approximate high performing view-dependent rendering scheme for parametric surfaces in this paper. With motion capture technologies become more prevalent, our scheme provides an ideal match for walk-around gaming applications. Our prototypes demonstrated the look and feel of such displays. The method, however, is only applicable to surfaces that can be ray-casted. More general surfaces are now finding ray-casting solutions [SN10] and can become suitable display candidates.We would like to explore rendering to more complex display shapes. Surfaces defined by splines can be used to approximate any surface and would provide a general solution to any display shape.

References

[BWB08] BENKO H., WILSON A. D., BALAKRISHNAN R.: Sphere: Multi-touch interactions on a spherical display. In *ACM User Interface Software and Technology* (2008), pp. 77–86.

[BWEN05] BIMBER O., WETZSTEIN G., EMMERLING A., NITSCHKE C.: Enabling view-dependent stereoscopic projection in real environments. In *International Symposium on Mixed and Augmented Reality* (2005), ISMAR '05, pp. 14–23.

[CNSD93] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A.: Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH* (1993), pp. 135–142.

[Dee92] DEERING M.: High resolution virtual reality. *SIGGRAPH Comput. Graph. 26*, 2 (1992), 195–202.

[DOS01] DJAJADININGRAT J., OVERBEEKE C., STAPPERS P.: Cubby: A unified interaction space for precision manipulation. In *Proceedings of ITEC 2001* (2001), ITEC 2001, pp. 24–26.

[FDO02] FRENS J. W., DJAJADININGRAT J. P., OVERBEEKE C. J.: Cubby+: exploring interaction. In *Designing Interactive Systems* (2002), DIS '02, pp. 135–140.

[Fis82] FISHER S.: Viewpoint dependent imaging: An interactive stereoscopic display. *SPIE* (1982), 367.

[HN09] HARISH P., NARAYANAN P. J.: A view-dependent, polyhedral 3d display. In *Virtual Reality Continuum and its Applications in Industry* (2009), VRCAI '09, pp. 71–75.

[Iwa04] IWATA H.: Full-surround image display technologies. *Int. J. Comput. Vision 58* (2004), 227–235.

[RWF98] RASKAR R., WELCH G., FUCHS H.: Seamless projection overlaps using image warping and intensity blending. In *Virtual Systems and Multimedia* (1998).

[SLF10] STAVNESS I., LAM B., FELS S.: pcubee: A perspective-corrected handheld cubic display. In *Human factors in Computing Systems* (2010), CHI '10, pp. 1381–1390.

[SN10] SINGH J. M., NARAYANAN P. J.: Real-time ray tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics 16* (2010), 261–272.

[SVF06] STAVNESS I., VOGT F., FELS S.: Cubee: a cubic 3D display for physics-based interaction. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches* (2006), p. 165.