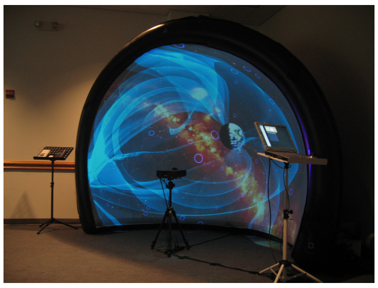
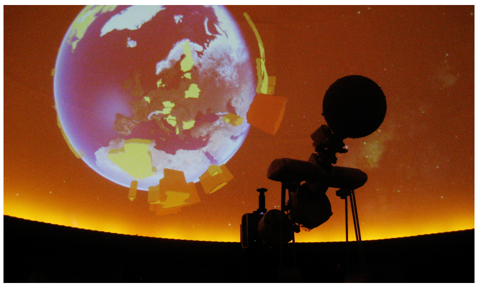
OmniMap: Projective Perspective Mapping API for Non-planar Immersive Display Surfaces

Clement Shimizu, Jim Terhorst, and David McConville

The Elumenati, LLC

Abstract. Typical video projection systems display rectangular images on flat screens. Optical and perspective correction techniques must be employed to produce undistorted output on non-planar display surfaces. A two-pass algorithm, called projective perspective mapping, is a solution well suited for use with commodity graphics hardware. This algorithm is implemented in the OmniMap API providing an extensible, reusable C++ interface for porting 3D engines to wide field-of-view, non-planar displays. This API is shown to be easily integrated into a wide variety of 3D applications.

1 Introduction

Artists, architects, and engineers have long attempted to understand and simulate spatial perspective as perceived by the human visual system. This quest is widely associated with the formal development of linear perspective from the 15th century onwards that became the hallmark of European Renaissance painting. These mathematically derived techniques forced an artificial projection of three-dimensional scenes onto a two-dimensional plane, requiring that the experience of perspective be based on the metaphor of peering through a planar window from a single point.

Numerous attempts have been made to define a "natural" non-planar and panoramic perspective that more closely mimics the visual field of view. Leonardo da Vinci illustrated the divergence between artificial linear perspective and a more natural non-planar perspective[1], and numerous artists have since turned to curvilinear and hemispherical architectural forms as canvases upon which to portray visual depth. Development of both optical and mathematical techniques for representing perspective within cathedrals, panoramic exhibits, planetaria, flight simulators, and surround cinema laid the groundwork for contemporary visually immersive display technologies[2, 3].

Though hemispherical and panoramic screens have long been used to surround audiences with imagery, their cost, complexity, lack of standards[4], and size requirements have limited their widespread adaptation. The use of panoramic and multi-screened virtual reality environments requiring multiple front and rear projectors have largely been limited to academic and corporate research and military training applications. However, recent advancements in graphics processing power, surround projection technologies, and material construction techniques are enabling the adoption of non-planar immersive virtual environments for a broad range of artistic, scientific, educational, and entertainment applications. Though many of the core technologies have been available for decades[5], a new generation of portable dome and panoramic systems as well as permanent digital "fulldome" theaters are providing relatively simple and cost-effective methodologies for visually immersing participants inside of computer-generated imagery using a single projector and image generator.

The OmniMap application programming interface (API) has been designed to enable existing interactive 3D software applications to be adapted for use within these wide field-of-view non-planar displays. It uses object-oriented techniques to enable extensibility, configurability, and reusability so that the minimum amount of effort is required for this adaptation. It is freely available on the web at [6] and has been integrated into a variety of 3D toolkits, programming environments and applications based on Direct3D9, Direct3D10, and OpenGL. This paper describes the algorithm employed to produce the correct viewing perspective output on non-planar display surfaces, the software architecture that supports the extensibility, configurability, and reusability, and the integration of the OmniMap API into various software applications.

The cover shot shows OmniMap integrated into SCISS AB's Uniview for realtime visualization of the universe (left). An OmniFocus projector sits alongside a star ball to update a planetarium's scientific visualization capabilities (right).

The next section provides an overview of various techniques used to produce imagery for non-planar immersive projection systems, including a two pass technique well suited for implementation on modern graphics hardware. Although various forms of the two pass algorithm have been used in many immersive computer graphics systems, the details have never been adequately described in the literature. This paper formalizes the algorithm in Section 3. Section 4 provides the implementation details of the OmniMap API needed to reproduce, utilize, and extend the technique. Finally, the flexibility and extensibility of the OmniMap API is evidenced by the successful integration into many applications and SDKs in Section 5.

2 Background

There are currently three primary approaches for rendering projections on non-planar surfaces: ray tracing, quadric surface transfer, and a two-pass rendering method. When projecting images onto non-planar display devices, ray tracing is the most straightforward approach. First, rays are cast from the projector to the screen's surface to compute the optical warping of the screen. From the viewer's position rays are cast through this intersection point into the scene to compute the final image. Because of computational requirements of raytracing, this only works for content that can be rendered offline.

Quadric surface transfer (QST) is a popular rendering technique to create large seamless curved displays using overlapping projectors[7–9]. QST systems typically need to use many projectors to cover a curved surface, rendering at very high resolution. The rendering phase for QST runs in a single pass, but that pass is run on each projector, so the implementation typically uses a computer for each projector. The greatest draw back of QST systems is the requirement for highly tessellated objects since major artifacts result if objects are not finely tessellated. Because either the application's 3D content needs to be re-tessellated offline or dynamically re-tessellated, QST is not yet practical as a general-purpose perspective correction library using commodity graphics processing units.

The two-pass rendering method is well suited for use with commodity graphics hardware since the scene does not need to be retesselated and is compatible with all versions of OpenGL and DirectX that support vertex and pixel shading. The first pass renders the scene into a set of off-screen textures, a method similar to generating cubic environment maps. The second pass renders and warps the screen surface and maps the textures to it using projective texturing. This is commonly used to correct perspective for immersive displays, including the OmniMap API. It has also been used within Elumens' SPIClops API for fisheye projection[10, 11], Paul Bourke's Spherical Mirror Projection[12], and the University of Minnesota's three-projector panoramic VR Window[13]. An early reference to a related two pass technique is found in [14].

As graphics processing power and projector resolution has increased, the two-pass rendering technique offers a method for taking advantage of simplified and lower-cost immersive projection systems. The OmniMap API has been developed to provide an extensible, reusable C++ interface that can be adapted to existing, and future commodity graphics hardware and most 3D software engines regardless of the type and level of abstraction. Although various forms of the two pass algorithm have been used in many immersive computer graphics systems, the details have never been adequately described in the literature. This paper formalizes the algorithm and provides the implementation details needed to reproduce, utilize, and extend the technique. In order to thoroughly describe the algorithms and implementations associated with the two pass rendering method, this paper proposes that it be named *projective perspective mapping.*

3 Projective Perspective Mapping

Typical video projection systems display rectangular images on flat screens. Optical and perspective correction techniques must be employed to produce undistorted output on planetariums, domes, panoramas, and other non-planar display surfaces. In this section we describe the methodology derived by the authors for *projective perspective mapping* which facilitates interactive perspective correct

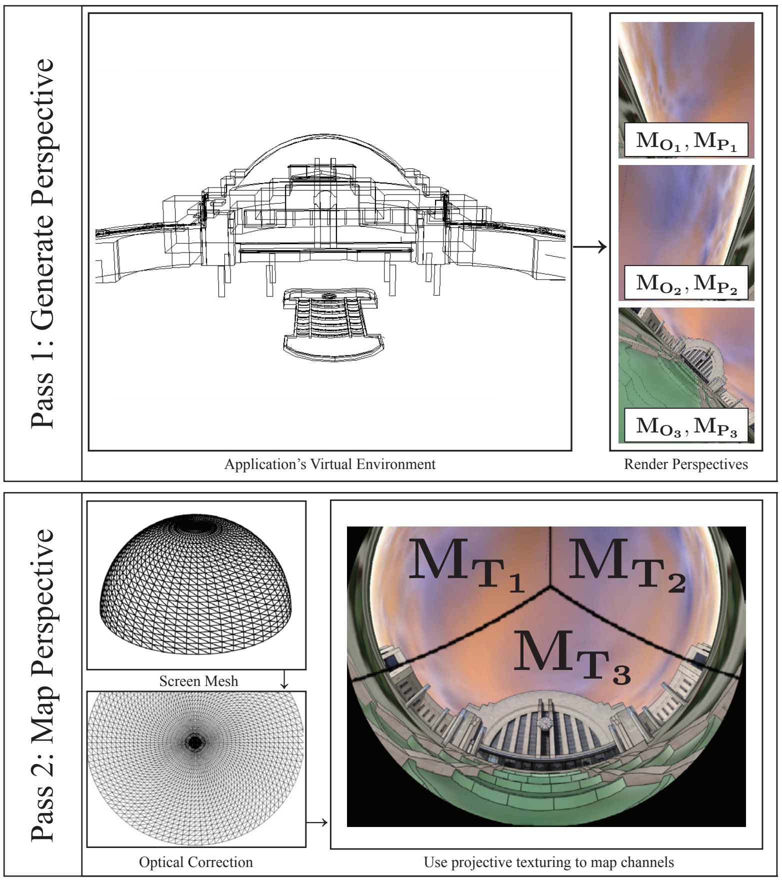


Fig. 1. *Projective perspective mapping* generates the user's perspective by rendering the scene into a subset of cube map faces, then uses projective texturing to map the perspective onto a optically corrected screen surface mesh. $\mathbf{M_{O_i}}$, $\mathbf{M_{P_i}}$, and $\mathbf{M_{T_i}}$ represent the view offset, projection, and projective texturing matrices for each channel

rendering into a wide frustum, non-planar surface. Although projective perspective mapping is capable of serving many projection system types, this discussion is focused on a specific class of projection systems with wide field of view optics similar to a fisheye lens.

Section 3.1 discusses the unique properties of the projector. Section 3.2 describes the algorithm's first pass where *channels* (cube faces) are rendered into off screen frame buffers (frame buffer objects or render textures). Section 3.3 describes the second pass, where a mesh representing the projection surface is rendered and warped using a vertex shader to account for the spherical projection of the wide field of view optics, and painted in using projective texture mapping. Figure 1 illustrates the flow of the algorithm.

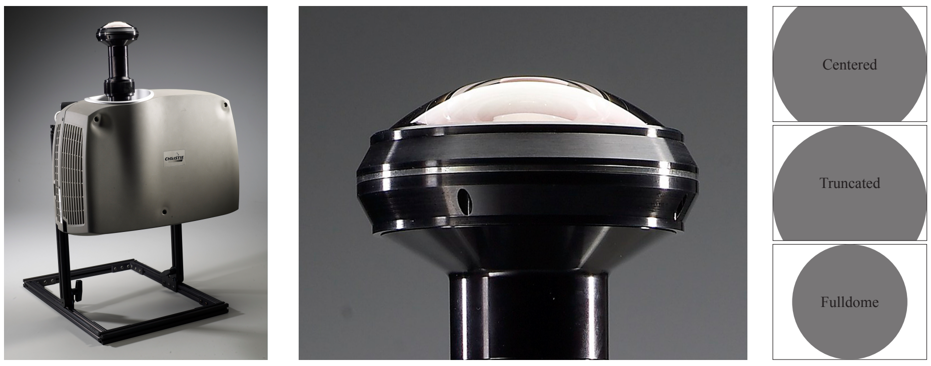


Fig. 2. A fisheye lens replaces a projector's stock lens. Lens offset configurations (right)

3.1 Projection System

The projector shown in Figure 2 is an Omnifocus^{TM}HAL-SX6 color projector with 6500 lumens brightness and 1400 x 1050 resolution. The stock lens from Christie has been removed and replaced with a custom wide field of view lens from The Elumenati, LLC. If the lens is centered on the projector's DMD panel, it projects 180° along the horizontal axis and a 135° along the vertical axis, Figure 2 (right, top). The lens is offset vertically relative to the center of the DMD panel to optimize pixel usage in projection. The resultant projected FOV is $\pm 90°$ horizontal and $+90°$, $-45°$ vertical (middle). "Full dome" lenses are available that project 180° in both the vertical and horizontal axis (bottom).

Most fisheye lenses tend toward an "$f\theta$" pixel distribution where the pixel projection angle is linearly related to the pixels distance from the lens optical axis. This means that the angle to which a specific pixel is projected is linearly proportional to its distance from the optical axis. The result is an equiangular pixel distribution across the entire projected field. Rectilinear projectors typically have an $f\tan(\theta)$ angular pixel distribution. In addition, if the $f\theta$ lens is centered in a dome, it will project with uniform brightness across the entire screen surface. If possible, avoiding brightness uniformity corrections is beneficial because they inevitably reduce screen's overall brightness and contrast.

While the extremely wide projected field of view allows a single projector to cover almost any screen shape, the nearly infinite depth of field allows the image to remain in focus. In optics a common rule of thumb is that for a lens of focal length f an object that is $\geq 20f$ away is essentially infinitely far away. Reversing this rule for projection, if the screen is $\geq 20f$ away from the lens the image will always be in focus. The fisheye used in this system has a focal length of 6mm. The extremely short focal length is a byproduct of the extremely wide FOV of the fisheye lens design. This allows the projector to be placed anywhere in relation to a screen of arbitrary shape (dome, cylinder, etc) and still maintain focus on the entire screen as long as the nearest point is at least 12 cm away.

3.2 First Pass - Generating the User's Perspective

Through the two-pass algorithm described next, projective perspective mapping takes into account the prescription of the optics described above, the shape of the screen and the position of the viewer within its algorithm in order to facilitate the generation of the correct image at the projector's image plate.

The initial pass generates the user's perspective by rendering the scene into faces (called render channels) of a cube-map like structure. A cube map with sufficient resolution can perfectly represent any view of the scene from a single vantage point. The scene is rendered through a subset of the faces of a cube. The n faces are chosen so as to fill the display surface with imagery from the perspective of the sweet spot of the audience. For each channel i, a view offset matrix $\mathbf{M}_{\mathbf{O_i}}$ is computed representing the offset from the default view to the view through the cube map face. This matrix also stores translational offset of the audience sweet-spot. The perspective projection matrix $\mathbf{M}_{\mathbf{P_i}}$ is also needed for each channel. The use of these matrices are the only change to the application's rendering loop. The scene's geometry does not need to be re-tesselated.

In the case of an upright dome, three channels are required to fill the display surface. The frustum and view offset rotation for each of those channels needs to be computed. If channel 1 is the left view, the matrix $\mathbf{M}_{\mathbf{T_1}}$ is a 45° rotation to the left. Channel 2's matrix $\mathbf{M}_{\mathbf{T_2}}$, the right view, is a 45° rotation to the right. The top channel's matrix $\mathbf{M}_{\mathbf{T_3}}$ would rotate and twist the view to capture the view through the top face of a cube. The perspective projection matrix $\mathbf{M}_{\mathbf{P_i}}$ for each channel is set to have a symmetric 90° FOVs and a near and far clip plane suitable for the scene. In the case where the optimal viewing position is not placed at the center of the dome, the ideal channel frustums may be asymmetric and greater than 90°. Asymmetric frustums enables an optimal frustum size, saving rendering time if the application is implementing frustum culling.

3.3 Second Pass - Mapping the views to the Display Surface

In the second pass, a mesh representing the projection surface is drawn with vertex and fragment shaders. First, the screen surface mesh is warped using the vertex shader, to account for the spherical projection of the wide field of view optics. Then, the channels rendered in the first pass are used as projective texture maps in the fragment shader to fill in the display surface.

Vertex Shader - Optically Correcting the Non-planar Display

Since the $f\theta$ lens causes the screen surface to be projected as spherically into the environment rather than rectangularly, the vertex shader is used to warp the screen surface from world space to spherical projection space. The screen mesh is tessellated to accommodate the warping. This allows the polygonal mesh of screen surface to be rendered in such a way that it lines up with the physical screen surface when projected through the spherical lens. This mapping is specific to the optics of the projection system. Although projective perspective mapping is capable of all types of projection systems, we work though the math for only the $f\theta$ optics. Calibration for rectangular projectors have been covered in [13] while mirror ball has been covered in [12]. This optical technique was published in the context of a rear projected, motion tracked, flexible screen in [15].

In the vertex shader, the world location of each vertex (x, y, z) is converted into spherical space (φ, r). z is defined as the axis parallel with the projector's direction. d is the distance between the vertex and the projector. In Equation 2, the vector φ is a unit length vector in screen space. Equation 1 computes its magnitude as r. Screen pixel coordinates (x', y') are computed by Equation 3. Finally, the z-depth is simply set to d.

$$r = \frac{2}{\pi} \cos^{-1}\left(\frac{z}{d}\right) \tag{1}$$

$$\varphi = \left(x/\sqrt{x^2 + y^2}, y/\sqrt{x^2 + y^2}\right) \tag{2}$$

$$(x', y') = r * \varphi \tag{3}$$

A slight modification needs to be made for fisheye lenses that have a have a non-uniform pixel distribution. To do this, a low order polynomial is fit to the mapping of the specific lens's distribution to an $f\theta$ distribution. The lens correction for standard, non-fisheye projector optics is covered in [13].

Fragment Shader - Mapping the User's Perspective to the Display

The fragment shader now fills in the display surface with the channels rendered in the first pass, using projective texturing. Projective texturing was invented as a technique for rendering shadows onto curved surfaces, but was extended to simulate the effect of a using a slide projector to project images onto curved surfaces[16]. A tech report on the subtle details of implementing hardware accelerated projective texturing can be found in [17].

The projection of the texture requires that the shaders calculate texture coordinates into the rendered channels drawn in the first pass for each pixel on the screen surface. These texture coordinates are calculated by applying a transform matrix to the vertex coordinate. For each channel i, the transform matrix $\mathbf{M_{T_i}}$ is computed from the projection and offset matrices used to generate the channel (Equation 4). The matrix $\mathbf{M_S}$ is a scale bias matrix for mapping coordinates from $[-1, 1]$ to the texture coordinate domain $[0, 1]$. The texture coordinates $(s/q, t/q)$ are computed for every channel by multiplying the vertex position (x, y, z) with the offset matrix (Equation 5).

$$\mathbf{M_{T_i}} = \mathbf{M_{O_i}} * \mathbf{M_{P_i}} * \mathbf{M_S} \tag{4}$$

$$\left(s, t, r, q\right)^{\mathrm{T}} = \mathbf{M_{T_i}} * \left(x, y, z, 1\right)^{\mathrm{T}} \tag{5}$$

If the screen surface is completely covered by the render channels from the sweet spot of the audience with no overlap, only one of the n texture coordinates will be valid for any pixel. The texture coordinates is valid if and only if (s_i, t_i) is within the range $[0, 1]^2$. The texture coordinate of the valid channel is used to index into the texture to retrieve the final color value for that fragment.

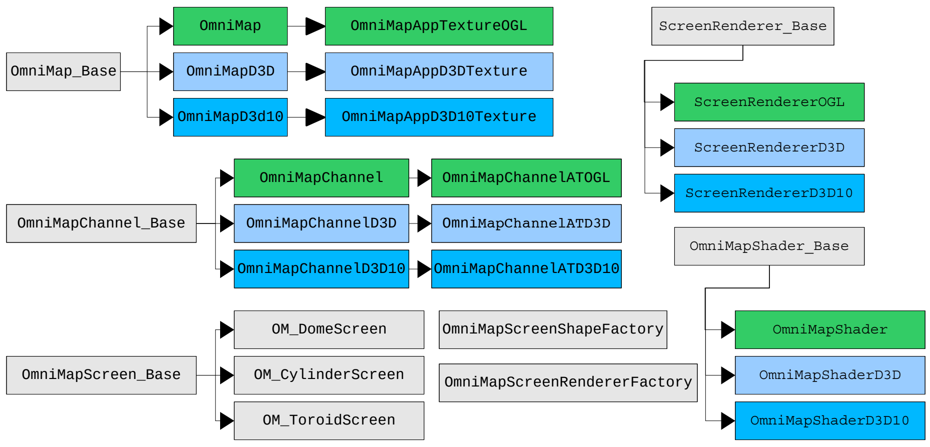


Fig. 3. Classes supporting specific graphics APIs like OpenGL (shown in green), Direct3D9 (light blue), and Direct3D10 (deep blue) are derived from graphics API independent (shown in gray) base classes. `OmniMap_Base`, `OmniMapChannel_Base`, `OmniMapShader_Base` and `ScreenRenderer_Base` need to be derived to incorporate the API specific function calls. For application supplied texture support, only main class, and channel classes needed to be derived. The shader and screen renderer classes were reused for the application supplied texture implementations. `OmniMapScreenShapeFactory` creates and manages `OmniMapScreen_Base` derivations, and `OmniMapScreenRendererFactor` creates and manages `ScreenRenderer_Base` derivations

4 OmniMap API Architecture

This section describes the OmniMap API architecture with an overview of the API and then a description of the functionality of the base classes and their derivations. The OmniMap API provides a framework to enable 3D application, game engine, and toolkit developers to implement rendering to immersive, non-planar displays. The base classes contain the information and utility access methods necessary to implement the algorithm. Derivations of the base classes provide the implementation details for specific rendering APIs and frameworks. The authors have derived the base classes to implement the algorithm for the OpenGL, Direct3D9, and Direct3D10 APIs, and some application architectures can use these implementations “out-of-the-box”. However, the extensibility of OmniMap affords developers the opportunity to adapt OmniMap to the framework of existing applications, toolkits or game engines. Toolkit and game engine developers can derive reusable classes from the OmniMap base classes, thus providing OmniMap functionality to application developers using their toolkit or game engine. OmniMap does not intend to provide the details of implementation for every conceivable software architecture, but instead provides the basic building blocks to enable developers to implement the algorithm with the least effort.

4.1 OmniMap API Classes

In this section, the support for the Projective Perspective Mapping algorithm in the OmniMap base classes is described, followed by an explanation of how these classes can be derived to implement the algorithm. The object oriented design of OmniMap enables runtime configurability of many properties of the display and software environment. These classes are designed such that derivations can implement the algorithm for:

1. Low *and* high level graphics APIs: OpenGL, DirectX, Orgre3D, etc.
2. Various shader languages, and shader loading/compiling APIs
3. Various screen shapes

OmniMap_Base: This is an abstract class for deriving classes that manage the two-pass algorithm. It owns the channel, shader, and screen objects. It directs the rendering of the second pass in which the channel content is composited to the display surface. It can be derived to implement the algorithm for different rendering engines, and has been derived to implement Direct3D and OpenGL. The base class provides functionality that is common to all derived implementations including:

1. Channel management
2. Creation, and storage of the matrix transforms that represent the position and orientation of the projector.
3. Screen shape factory[18], screen management, screen renderer factory, screen renderer management, and invocation of screen rendering.
4. Utility methods for calling the list of channels to bind and unbind their respective textures. This method is used during the second pass to bind the channel textures to texture units for access by the shader.
5. Utility methods for executing Lua scripts, which are used primarily for configuration.

`OmniMap_Base` has two object factories, one for the creation of screen shapes (see `OmniMapScreen_Base` below), and one for the creation of screen renderers. These factories allow for new screen shapes, and screen renderers to be created and added at run time.

`OmniMap_Base` uses a Lua scripting facility to initialize itself with the preferred configuration[19]. The Lua script file is executed by the base class. The script then calls back into the `OmniMap_Base` class through a Lua to C++ interface mechanism[20]. These calls provide the OmniMap object with configuration information including:

– Number of channels to be rendered, their resolution, and projection parameters that define how those channels are composited onto the display surface.
– Underlying Graphics API to be used for rendering
– Shader programs to be used for vertex warping and channel projection
– Position and orientation of the projector, and optimal viewing position

OmniMap includes configuration scripts for standard dome configurations shipped by Elumenati, Inc. The Lua scripting facility is available to derived classes via a protected member of each base class.

The OmniMap API base classes are free of any code that is graphics API dependent. The design allows implementations of the perspective mapped surface display algorithm for high (Ogre3D, OpenSceneGraph) and low level graphics APIs (OpenGL, Direct3D9 and Direct3D10) to leverage the base functionality, minimizing the efforts required for those implementations. Derived implementations of `OmniMap_Base` must implement the `CreateChannel` and `PostRender` methods. The `CreateChannel` method creates the appropriate derivation of `OmniMapChannelBase` for the derived implementation. `PostRender` implements the second pass of the algorithm Specifically, shader loading and parameter setting, and display surface rendering.

The OmniMap API also provides support for applications that supply their own render textures. This support is useful for toolkits that already have a facility for rendering to textures. For instance, `OmniMapAppTextureOGL`, and `OmniMapChannelATOGL` implement this functionality for OpenGL applications.

OmniMapChannel_Base: This is the abstract class for channel implementation. It is intended to be derived along with `OmniMap_Base` to support a specific rendering API or framework. Its purpose is to provide the mechanism for rendering to an offscreen buffer that acts as a texture, and to provide the matrices for setting up the channel's viewing offset. Hence, this class owns the position/rotation offset matrix and the projection matrix for the channel. The `OmniMapChannel_Base` class supports asymmetric frustums.

OmniMapScreen_Base: This is the abstract class for implementing display surface shapes. Derived classes simply define vertex buffers that represent the geometry of the display surface. The screen geometry is rendered with classes derived from the `ScreenRendererBase` class. This class is defined such that the derived classes can be implemented independent on the underlying graphics API. So, a screen shape can be defined once, and used by any derivations of the OmniMap classes, as well as any game engine or toolkit implementations. Derivations that implement specific screen shapes are responsible for tessellating the shape into triangles and notifying the screen renderer of the contents of that shape.

ScreenRenderer_Base: This is the abstract class for defining screen renderers. Screen renderers are responsible for rendering the shapes defined by derivations of `OmniMapScreen_Base`. Derived classes are graphics API dependent. So there are derived classes for OpenGL, Direct3D9, and Direct3D10 rendering. The derived classes simply render the vertex buffers defined by classes derived from `OmniMapScreen_Base` class.

OmniMapShader_Base: This is the abstract class for implementing shader support. Derived classes are responsible for loading/unloading, compiling, and setting parameters in shaders.

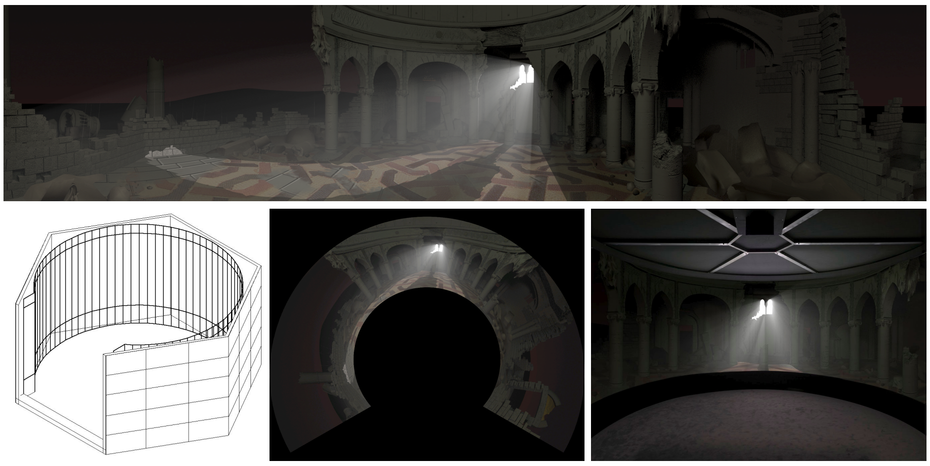


Fig. 4. Michael Somoroff's *Illumination* features a downward-facing projector filling a 120° wrap-around panoramic display. A custom video player was created using OmniMap to take the high resolution panoramic frames (top) and warp them for projection onto the inside of a 120° cylindrical display (bottom left). OmniMap produces a dramatically warped but optically correct horseshoe shaped image (bottom middle) that is projected into the final installation (bottom right)

5 Conclusion

As advances in graphics processing power, surround projection technologies, and material construction techniques enable proliferation of non-planar immersive display devices, demand for content accelerates. The authors conclude that a library for integrating the wide variety of existing interactive 3D software applications into these venues is a necessity and should be freely available. The OmniMap API is free for non-commercial use, runs on OSX and Windows. The OmniMap API provides a simple way for software developers to implement rendering into non-planar immersive display surfaces. This is evidenced by the successful integration of OmniMap into the following applications and SDKs:

- Applications
  - GeoFusion, Inc. GeoPlayer : Geospatial Visualization
  - Google's Sketchup : 3D Modeling.
  - SCISS Uniview : Astronomy Visualization
  - Apple's Quartz Composer : Interactive, Visual Programming Language
  - Linden Lab's Second Life Viewer : On Line Virtual World
  - VidVox's VDMX : Realtime Video Mixing and Effects Software
  - Cycling '74's MaxMSP : Realtime Video Mixing and Effects Software
  - Elumenati Video Player : High Def Codec for mono/stereo videos (Fig. 4.)
- SDKs
  - OpenSceneGraph 3D Toolkit
  - Unity 3D Game Engine
  - Ogre 3D Game Engine

Recently, the authors have successfully used OmniMap to implement active and passive stereo systems as well as multiple projector configurations all powered by a single PC, and plan to integrate general implementations of these functions into the library. The many developers bringing exciting applications to immersive displays using OmniMap prove how projective perspective mapping systems are elegant, simple, and cost effective in their computer hardware, projector optics, and software implementation.

References

1. Blake, E.H.: The natural flow of perspective: Reformulating perspective projection for computer animation. Leonardo **23** (1990) 401–409
2. Benosman, R., Kang, S.: A brief historical perspective on panorama. Panoramic vision: sensors, theory, and applications (2001) 5–20
3. McConville, D.: Cosmological cinema: Pedagogy, propaganda, and perturbation in early dome theaters. Technoetic Arts **5** (2007) 69–85
4. Lantz, E.: A survey of large-scale immersive displays. In: EDT '07: Proc. of the 2007 Workshop on Emerging Displays Technologies, ACM (2007) 1
5. Shaw, J., Lantz, E.: Dome theaters: Spheres of influence. In: Trends in Leisure Entertainment. (1998) 59–65
6. The Elumenati, LLC: Omnimap api, real–time geometry correction library. http://www.elumenati.com/products/omnimap.html (2008)
7. Raskar, R., van Baar, J., Willwacher, T., Rao, S.: Quadric transfer for immersive curved screen displays. Comput. Graph. Forum **23** (2004) 451–460
8. Raskar, R., van Baar, J., Beardsley, P., Willwacher, T., Rao, S., Forlines, C.: ilamps: Geometrically aware and selfconfiguring projectors. SIGGRAPH (2003)
9. Majumder, A., Brown, M.S.: Practical Multi-projector Display Design. A. K. Peters, Ltd., Natick, MA, USA (2007)
10. Elumens Corporation: The SPIClops API. (2001)
11. Chen, J., Harm, D.L., Loftin, R.B., Lin, C., Leiss, E.L.: A virtual environment system for the comparison of dome and hmd systems. In: Proc. of the International Conference on Computer Graphics and Spatial Information System. (2003) 50–58
12. Bourke, P.: Low cost projection environment for immersive gaming. In: JMM (Journal of Multimedia). Volume 3. (2008) 41–46
13. Ries, B., Colucci, D., Lindquist, J., Interrante, V., Anderson, L.: VRWindow: Tech Report. Digital Technology Center, University of Minnesota. (2006)
14. Greene, N., Heckbert, P.: Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. In: IEEE Computer Graphics and Applications. (1986) 21–27
15. Konieczny, J., Shimizu, C., Meyer, G.W., Colucci, D.: A handheld flexible display system. In: IEEE Visualization. (2005) 75
16. Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., Haeberli, P.: Fast shadows and lighting effects using texture mapping. SIGGRAPH **26** (1992) 249–252
17. Everitt, C., Rege, A., Cebenoyan, C.: Hardware shadow mapping. Technical report, http://developer.nvidia.com/object/hwshadowmap_paper.html (2002)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts (1994)
19. Ierusalimschy, R.: Programming in Lua, 2nd Edition. Lua.Org (2006)
20. Schuytema, P., Manyen, M.: Game Dev. with LUA. Charles River Media (2005)