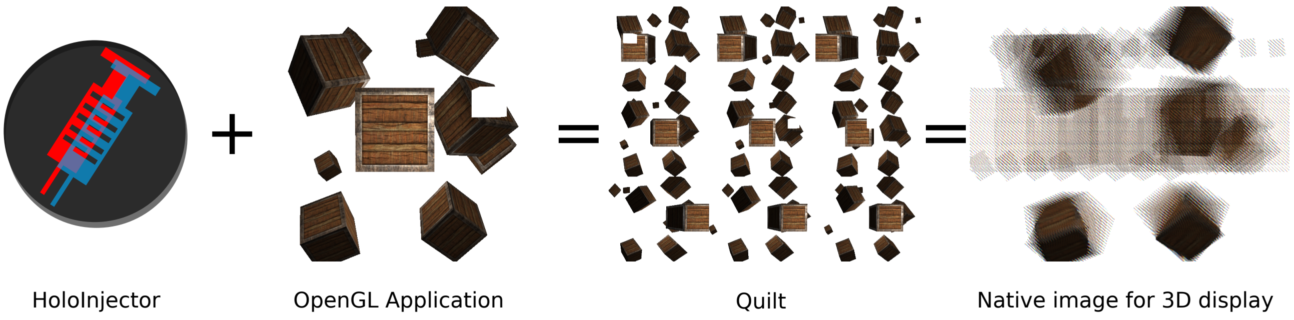
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Holographic Injection - Let There Be True 3D

Bc. Roman Dobiáš*



Abstract
The adaptation of upcoming autostereoscopic displays by regular users depends on availability of supported applications. To increase such set, this paper describes compatibility software which turns (semi-) automatically the output of generic OpenGL 3D application to display-native output, while taking advantage of true 3D display capabilities. This is achieved using a conversion layer that intercepts parts of OpenGL API and translates such API calls to others to produce multiview output of the original application.

Keywords: OpenGL, autostereoscopic displays, single to multiview conversion, automated conversion, pipeline injection, API call hooking

Supplementary Material: Downloadable Code, Examples of conversion

*xdobia11@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

There are tons of legacy applications in the real world that would be interesting to display using autostereoscopic displays (shown in Figure 1) while taking advantage of their depth perception capabilities. However, the conversion of applications usually requires significant amount of additional work for developers to adapt the rendering part of engine. In addition, the development of many applications has already deceased, or the source code could be lost.

This paper attempts to solve this by introducing an application-agnostic compatibility layer, placed between the application and underlying OpenGL driver. It transforms OpenGL API calls to different API calls transparently so that the output of the original application is turned to the display-native image format of desired autostereoscopic display in run-time, as shown in Figure 2. The implemented software is limited to OpenGL applications, but DirectX could be supported analogously.

Figure 1. Consumer-affordable autostereoscopic display, produced by Looking Glass Factory Inc. [1]

In theory, such the layer could convert any OpenGL application, but in practice, this is unfeasible due to large and complex state space of OpenGL API calls and their respective valid applications. Moreover, some of the effects used in rendering may not be reproduced without additional knowledge due to ambiguities. Therefore, the practical output of this paper is a tool which helps to semi-automatically convert a subset of applications with minimum guidance by user.

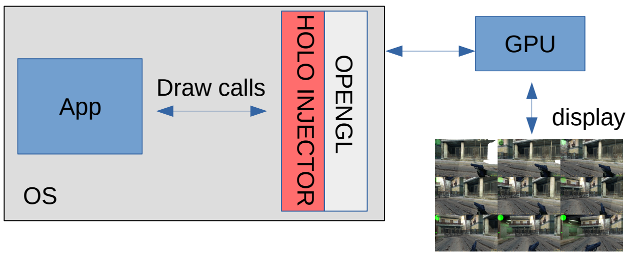


Figure 2. Resulting conversion layer, placed in operating system. Draw calls to OpenGL API are intercepted and duplicated transparently, resulting in multi-view output of the original application.

Up to my knowledge, existing solutions are limited to conversion of single- to stereoscopic-view displays such as Head-Mounted Displays for Virtual Reality, or synthesize multiview from stereo view.

The paper's contribution is in describing the process of extending single-view 3D applications to autostereoscopic-ready formats, in providing an implementation of solution working for a subset of all possible existing OpenGL applications, and in showing that it is possible to partially handle this task automatically.

2. Background

Before describing the solution, a prerequisite knowledge of displaying using autostereoscopic displays is briefly described.

2.1 Autostereoscopic displays

In contrast to Stereoscopic Displays, which provides two distinctive views of scenes, such as anaglyphs or Head-Mounted Displays, *Autostereoscopic Displays* do not require user to wear or use any additional gadgets.

The most prominent ones are *Light-field Displays*. These provide different image, depending on angle under which the display is perceived [1, p. 16] with limited angular density (unique views n, typically $n = 45$). In the remaining text, we assume unique views only in horizontal plane of the display.

They are typically made of a single LCD screen with an additional element in front, *parallax generator*, which effectively blocks the visibility of pixels belonging to other views under a certain angle. In production displays, an array of micro-lenses called *lenticular lens* are used. As lens is typically spread in a diagonal direction to better use the resolution of underlying screen, such lens is referred as *slanted lens*.

Alternatively, volumetric or holographic displays are also referred as autostereoscopic, but they operate over different representations of a scene.

2.2 Image structure of display's native format

The physical layout of pixels belonging into logical views is referred as so-called *native format* of 3D display, and it depends on physical measurements of display such as angle of slanted lens, size of lens, etc.

To provide a universal interface between views of scene and displayed bitmaps, so-called *quilt* [2] image format is introduced, represented by a grid of views. Each cell is a view with well-defined camera transformation, going from the leftmost view (bottom-left cell) to the rightmost view (top-right cell). The quilt is then transformed with display-specific parameters into the resulting display-native image, which is shown on the embedded screen of the display. An example of quilt and its corresponding native image is shown in Figure 3.

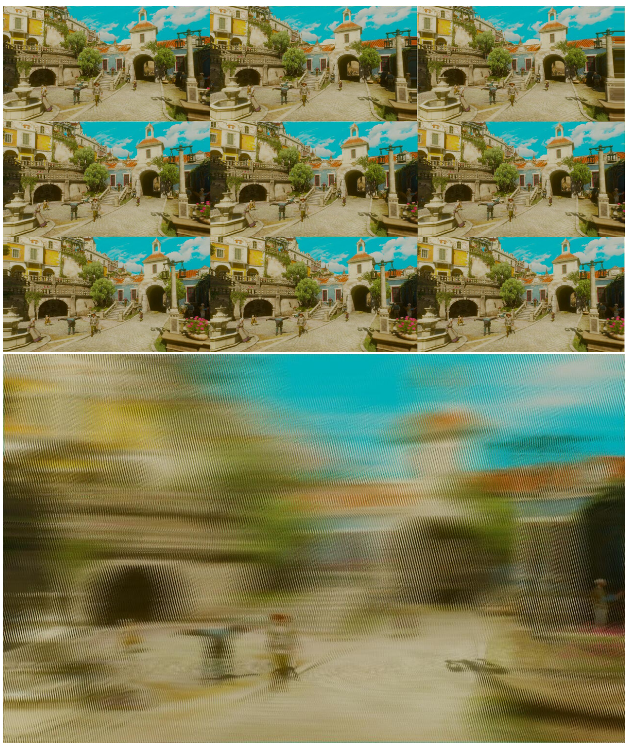


Figure 3. An example of quilt - a 3x3 grid of views, manually taken from game The Witcher (top), and the result of transformation to native format (bottom).

a In the subsequent text, a view refers to a cell of quilt if not mentioned otherwise.

2.3 Preparing content for autostereoscopic displays

A typical autostereoscopic application thus renders the scene from different perspectives in each frame, and either feeds the quilt to display-specific SDK (for instance, *HoloSDK* [3]) or outputs the native image directly.

2D applications can be converted by rendering the same view to all cells of quilt.

To make use of 3D perception with traditional applications, applications have to be reprogrammed so that the scene is rendered from different views close to the original one, which may be resource-demanding.

This paper attempts to provide an alternative solution for the aforementioned porting cases by introducing a conversion layer which translates rendering commands to multiview rendering without changing the code of original application.

2.4 Existing methods & applications

To my best knowledge, there is neither academic nor open source solution for automated or semi-automated conversion of real-time 3D applications to autostereoscopic displays. The closest type of similar existing application is a conversion of generic 3D application to binocular headsets, used by VR. In that field, numerous commercial and open-source applications exist.

In general, there are *two approaches for single to stereo conversion*. The most generic approach is based on estimating stereo output approximately by post-processing the application's output RGBD image. This is typically done by using methods derived from *steep parallax mapping*, in which depth buffer stands for local surface of geometry. Besides, these methods must tackle solving holes, stemming from depth buffer discontinuity, and may require user-defined bitmap masks of pixels defining HUD. Such method is implemented in *Depth3D* [4], an extension of *ReShade* framework, or in commercially-available vorpX [5]. This approach satisfies needs of VR, assuming a short baseline between user's eyes, but would cause massive visual artifacts when pushing disparity to longer distances, due to missing information. Whereas typical human has 50-60mm long baseline [6] between eyes, users of autostereoscopic display may watch the display from distance comparable to LCD displays, resulting in large horizontal baseline. An example of such visual artifact is visible in Figure 4.

Secondly, software such as *vorpX* also tries to inject rendering pipeline and directly renders the application's content twice with different transformations. However, none of existing commercial software reveals any formal description of this process. Also, many solutions provide lists of supported applications[2] suggesting that the underlying method does not work transparently, but instead requires application-specific tweaks or patches.

[2] https://www.vorpx.com/supported-games/

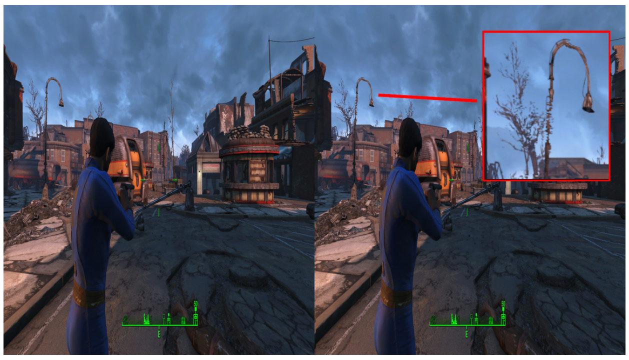


Figure 4. Visible visual artifact along edges in output of vorpX, when used for VR, suggesting use of steep parallax mapping for view duplication.

Alternatively, numerous methods exist in field of conversion from *stereo to multiview conversion* [7] by estimating depth map of from two views using disparity and subsequent reprojection or warping of input images.

The state-of-the-art method [8] can successfully reconstruct novel views and extrapolate input views to provide fill the whole quilt. However, any reconstruction method will always be limited by amount of information, provided in two input views, and therefore, in case of obstruction of the scene by a close object, the resulting views may lack the geometry behind the obstructor. While this may be sufficient for film industry, in general, interactive applications try to minimize artifacts as these might be explored by user purposely, resulting to the break of the illusion.

In conclusion, the proposed conversion layer can provide ground truth views into the scene even for obstructed scenes, which could be the potential advantage over reconstruction methods.

3. Proposed solution

In general, graphical applications communicate with underlying hardware in terms of geometry calls, which upload representation of 3D model to graphic card, and *draw calls*, which command the card to render the models using specific parameters.

The ultimate goal of conversion layer is to intercept and duplicate the draw calls with altered transformation so that meshes would be drawn multiple times into multiple views, as shown in Figure 5.

The goal is achieved using following steps:

1. **Shader injection** During link time, each shader program, used for rendering, is adapted such that the transformation pipeline would allow to render different views of quilt, based on parameters

in uniforms. This includes extraction of projection matrix out of *ModelViewProjection*(referred as `MVP` latter) matrix, and reprojection after altered transformation.
2. **Uniforms tracking** Uniforms are tracked to obtain current `MVP` matrix when it is passed to GPU using uniform API calls. Intercepted matrix is subsequently decomposed.
3. **Draw call duplication** Each draw call is intercepted and dispatched multiple times, each for corresponding view. The correct view is obtained using extracted projection parameters to revert *clip-space* output of *Vertex Shader* (referred as VS) to *camera-space*, shift the position, and reapply estimated projection.
4. **Post-processing** At the end of each frame, the obtained quilt is post-processed to the native image.

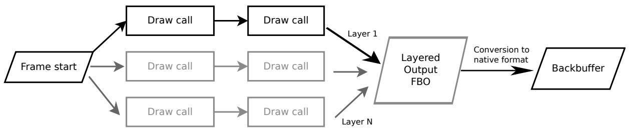


Figure 5. A generic approach to conversion: Draw calls are intercepted and duplicated internally for each view of quilt. Rendering to the Back buffer is detected and replaced with rendering to layered FBO, representing the quilt. In the end of frame, the quilt is post-processed and rendered to backbuffer.

3.1 Injection to programmable pipeline

Shaders and programs are changed by intercepting their respective API methods for creation, attachment, compilation, and linking. It's sufficient to intercept linking itself, and use OpenGL querying methods for introspection of attached shaders to obtain their source codes.

In order to draw multiple views, it's necessary to alter VS or GS so that the transformation of vertices is changed for corresponding view. OpenGL's standard itself does not define how the transformation should be arranged in shaders for input data. Applications are free to set up arbitrary sequence of statements, which fulfill their needs, provided the chain results in output vertex position in *clip-space*.

In this paper, we skip description of injection to *Geometry Shader* (referred as GS), as it is done analogously with only few additional steps.

The transformation shaders in most of applications may have one of following variations:

- **Identity function**
  Vertex shader simply copies the input vertex data to output. This kind of behavior is used for instance for rendering full-screen quad geometry for post-processing, or for rendering elements of GUI, which tends to have their position precalculated at CPU-side.
- **Matrix multiplication**
  The most common shader type, which follows `MVP` transformation. Typically, it gets either `MVP` or pair *Model* and *View-Projection* matrices using uniforms, and then, simply multiply the input vertex position with these matrices.
- **Far plane rendering**
  Far plane rendering is a special case of previous methods, in which the resulting output vector has w component set to 1, effectively placing the geometry at the position of far plane. In addition, this maybe accompanied with *Model-View* matrix only using rotation and scale to simulate directional rendering.
  This is commonly used when rendering skyboxes.
- **Constant propagation**
  Vertex shader sets output position to a hard-coded vector, stored inside shader's code. This is an extreme case, for instance, used to generate a full-screen quad.

By determining which variant the vertex shader implements, it's possible to identify the projection matrix's uniform name, which is need for reverting *clip-space* to *camera-space* position. If the transformation shader does not use matrices, the alternation of shader is terminated, and the original shader program is used. This typically happens for HUD rendering.

The injection thus follows two goals: finding the name of projection matrix and altering the transformation pipeline to append reversion, shift, and reprojection at the end of shader. The latter is described in Section 3.3. The resulting pipeline is visualized in Figure 6.

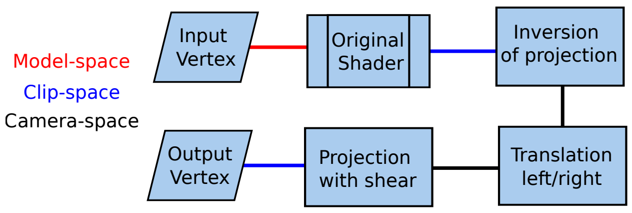


Figure 6. Steps in new transformation shader, created using composition of the original shader and additional steps. Conversion of VS is possible by renaming main() function and introducing new main() function with additional steps.

In theory, analysis is done by parsing GLSL and inspecting the resulting Abstract Syntax Tree. Due to the shortage of time, it was instead implemented using *Regular expressions* over source codes. In both cases, the analysis starts with tracking operations, used for assigning to `gl_Position`.

After deciding the type of transformation of original VS or GS, it is necessary to remove projection from the transformation to obtain output positions in camera-space instead of clip-space. Then, the horizontal translation for corresponding views is applied, and the original projection can be reapplied.

In the general case, it is necessary to estimate projection from a single `MVP` matrix. The following section describes when such decomposition is possible.

3.2 Estimating projection from ModelView-Projection matrix

It is possible to show that projection matrix can be extracted from `MVP` provided that projection is symmetric projection and *Model-View* matrix (referred as `MV`) only contains so-called *uniform scaling matrix* (thus, equal scale in all dimensions) as a scaling matrix. This isn't always necessary as programmers prefer splitting `MVP` to model and view-model parts to minimize uniform-passing operations and CPU cycles.

Instead, we give the idea of extracting projection parameters from *View-Projection* matrix (referred as `VP`). In generic case, *View* matrix (referred as `V`) defines inverse of camera transformation in world-space, and thus, it can be expressed as some $R \in SO3$ and $t \in \mathbb{R}^3$.

Symmetric projection with parameters n (near-plane distance), r (right clip-plane), t (top-clip plane) and f (focal length) is defined in Equation 1.

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{1}$$

Let's denote $F_x = \frac{n}{r}$,$F_y = \frac{n}{t}$, $A = \frac{-(f+n)}{f-n}$ and $B = \frac{-2fn}{f-n}$. By multiplying V with P, obtained `VP` matrix has the property of $norm(x_{11}, x_{12}, x_{13}) = F_x$ due to orthonormality of $SO3$ and diagonal form of 3×3 submatrix of projection. The same property applies to row 2 and 3, thus we can obtain F_x,F_y and A directly by normalizing 1th, 2nd and 3rd row of 3×3 submatrix of the resulting matrix, respectively.

Remaining B can be obtained by solving linear system $A \cdot (-x_{44}) + B = x_{43}$.

3.3 Injecting transformation's chain

Obtained parameters are sufficient to transform *clip-space* coordinates to *camera-space* by multiplying x and y with inverse of F_x and F_y, respectively. Depth can be extracted as $-w$. Afterwards, the vertex is translated and projection is applied by constructing projection using extracted parameters and shear for side views of quilt. Note that *clip-space* coordinates in OpenGL are homogeneous coordinates prior to division by w component, and thus, such operations are possible.

Also, note that projection parameters are only extracted when a new matrix is uploaded to shader via `glUniform4v`. The extracted parameters are then passed to each draw call as an additional *vec*4 uniform.

3.4 Shadowing of Frame Buffer Objects

The aforementioned conversion works fine provided the application renders directly to the Back buffer. As complex applications use artificial *Frame Buffer Objects* (referred as FBO), it is necessary to define the flow for these. One approach is to split the texture of FBO to a grid directly and use `glViewport` internally, but this requires tracking to detect if the texture is sampled during the next draw calls, and intercept & alter sampling code in shader so that proper subregion of texture is sampled instead.

Alternatively, it is possible to replace (shadow) an FBO with an internally created layered FBO, made of layered textures. The number of layers matches the count of views in quilt. This approach requires tracking of FBO's lifetime and replacing sampling operations of the layered FBO's texture in all affected shaders. This paper relies on this approach.

In conclusion, during each draw call, the layered FBO is rebound and the draw call is drawn using *Geometry Shader* with instancing. If such drawing is possible only using `VS`, a temporary FBO is created for each view of quilt of given layered FBO. This possible thanks to *Texture View* mechanism of OpenGL. A texture view is a proxy texture object which may point to sublayer or sublevel of a different texture.

Finally, when the texture of the original FBO is about to be rendered, the intercepted draw call is dispatched for each view with view's Texture View, bound instead of the texture.

4. Fixed-pipeline rendering

For injection to fixed-pipeline, it is sufficient to track draw calls as the transformation only depends on the top of the transformation stack, which is filled by application using specialized API calls. The translation of side views can be added by multiplying the top of the transformation stack (`GL_MODELVIEW`) from left using `glMultMatrix` with translation prior to each duplicated call. The projection is always stored in a separate GL_PROJECTION stack and can be affected and investigated by tracing operations for pushing matrices to the stack.

4.1 Replicating geometry using glCallList

A typical fixed-pipeline application uploads the geometry by successive sequential API calls, vertex by vertex. In theory, in order to replicate such mesh, one would have to record the sequence of geometry uploads and replicate the sequence each time a different view is rendered.

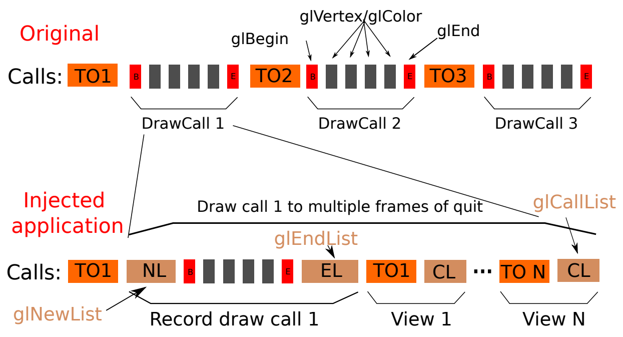


Figure 7. An OpenGL calllist is used to record sequence of geometry API which made up a draw call. Subsequently, the draw call is repeated for each view of quilt by calling the list.

Fortunately, such mechanism is already implemented in OpenGL under name glCallList. A *call list* is a recorded sequence of OpenGL API calls, supporting a subset of OpenGL API, mostly including geometry, transformation and shading API calls. The use of call list for duplicating draw calls is illustrated in Figure 7.

5. Implementation

The method described in this paper has been implemented for OpenGL on Linux platform. The conversion layer is compiled as a dynamic library, which is injected into the application's process using LD_PRELOAD functionality and methods are hooked by attacking dlsym/dlopen functions to override addresses of imported symbols.

In addition, hooked functions are also defined and exported explicitly with linked shared library to support applications which are statically linked to a shared library of graphical driver.

6. Limitations

The limitations of the presented method can be divided into two categories. Firstly, problems stem from ambiguities and limited knowledge of black box conversion. Secondly, due to limited time to deliver, implementation trade-offs are causing failure in a few specific use-cases.

6.1 Flatness of HUD

Any rendering technique which projects 3D positions to screen and pass such position to transformation instead of transformation matrix itself are limited to flat 2D rendering due to missing spatial information. Naturally, this is mostly the case for HUD, but in addition, this may affect billboarding as well.

6.2 Frustum culling

The presented method works over subspaces of volume, provided in draw calls. A typical optimized engine will employ techniques to skip drawing of meshes, which are invisible from the application's point of view. This may affect side views of quilt due to missing information in draw calls.

6.3 Shading transformations

Applications implement shading in fragment shader based on angles or positions of lights and position of view. In order to achieve shading which corresponds to the correct shifted position of view, these must be altered as well. However, as no standard exists for passing such data to fragment shader, more complex analysis is required to understand which outputs of transformation pipeline must be changed, and this typically fails.

This limitation is clearly visible when rendering reflective materials, resulting in improper specular reflections, and can be perceived in Figure 8.

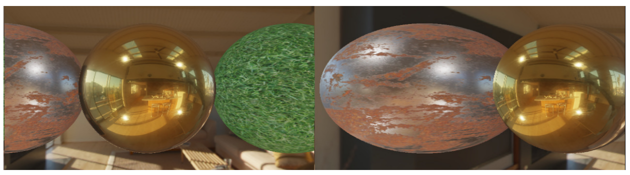


Figure 8. The same reflection on golden sphere's surface in side view (right) as in the front view (left), caused by missing propagation of altered transformation matrix to computation of camera-space position and pixel's normal in Fragment Shader.

This could be solved by allowing experienced users to manually edit the injected transformation shader. Changes could be associated permanently with specific shader by hashing content of the original shader.

6.4 Complexity of programmable shaders

Currently, *Regular expressions* are used to automatically extract metadata about used uniforms and operations in shaders. This results to failures of detection in complex applications, which may use variable shadowing or if-else branching. We believe this could be improved by using a proper GLSL parser and employing more complex analysis.

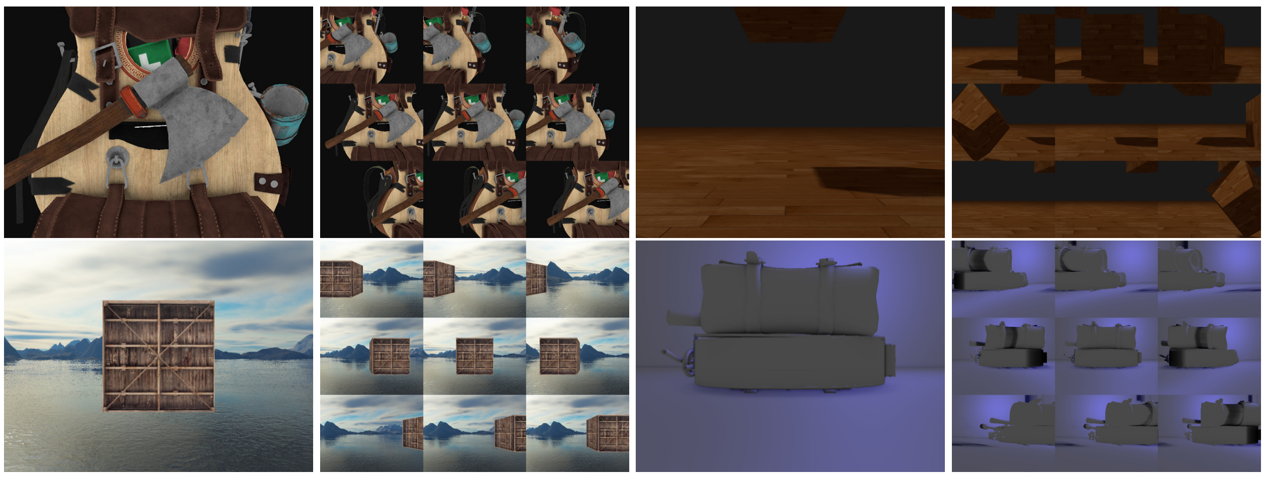


Figure 9. Pairs of images, showing original application (left) and resulting quilt (right). The first pair demonstrates ability to convert a regular render with more complex mesh. The second pair demonstrates shadow mapping consistency. The third features skybox rendering. The last sample shows failure due to technique SSAO, which uses transformations in Fragment Shaders. All examples were converted automatically, and originate from *LearnOpenGL*'s repository.

Q/Res	128^2	256^2	512^2
1x1	16.5ms	16.5ms	16.5ms
3x3	63.8ms	70.3ms	71.5ms
5x9	263.9ms	271.0ms	284.5ms

(a) Stanford dragon (complex geometry)

Q/Res	128^2	256^2	512^2	1024^2	2048^2	4096^2
1x1	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms
3x3	16.5ms	16.5ms	16.5ms	20.6ms	36.0ms	56.4ms
5x9	19.4ms	32.3ms	50.2ms	69.5ms	93.1ms	166.5ms

(b) Steep Parallax Mapping scene (complex shading)

Q/Res	128^2	256^2	512^2	1024^2	2048^2
1x1	16.1ms	16.2ms	16.1ms	16.5ms	16.4ms
3x3	16.5ms	16.5ms	16.5ms	16.5ms	16.5ms
5x9	16.5ms	16.1ms	24.2ms	37.9ms	65.6ms

(c) Cubes (simple geometry)

Q/Res	128^2	256^2	512^2	1024^2	2048^2
1x1	21.3ms	22.5ms	21.0ms	21.3ms	21.8ms
3x3	23.4ms	22.1ms	26.0ms	25.8ms	32.7ms
5x9	67.3ms	74.5ms	86.0ms	105.3ms	144.3ms

(d) Asteroids (many trivial draw calls)

Table 1. Average frame period. Lower is better. Quilt (number of views) vs resolution (width/height of each of view). Test setup: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz and NVIDIA GTS 920M.

7. Experiments with applications

The implementation has been tested on publicly available OpenGL applications.

The programmable pipeline was tested using OpenGL tutorials such as *LearnOpenGL*[3] to verify different techniques. The method has been employed on examples of techniques such as Shadow mapping, Normal mapping, Skybox drawing, or drawing to auxiliary framebuffers. For techniques such as Screen-Space Ambient Occlusion (SSAO), the method failed to provide consistent multiview image, due to suspected problems in passing data to FS (see Section 6.3).

The fixed-pipeline OpenGL was tested on such as *NeHe tutorials*[4] and more complex application *Nexuiz*. Due to simplicity of fixed-pipeline transformation, an implementation based on the method above is sufficient to convert most of such applications with minimal visual drawbacks.

[3]GitHub.com:JoeyDeVries/LearnOpenGL

[4]GitHub.com:gamedev-net/nehe-opengl

7.1 Performance measurement

The impact of injection was measured on selected supported applications. The results are shown in Table 1. Each measurement was a recording of frame period (time per frame) in milliseconds. The measurements test correlation between the frame rate and increasing number of views, and correlation between increasing resolution and frame period.

Three different applications were used. *Stanford dragon* features a large mesh, which should solely fill the pipeline. *Cubes* provides trivial geometry with simple shading. The purpose of *Steep Parallax Mapping* is to feature simple geometry, but expensive shading, so that the dependency on resolution could be measured. Finally, *Asteroids* provide hundreds of draw calls with trivial geometry

Clearly, it can be concluded that the injection and rendering into multiple views affect performance. In case of complex geometry, but cheap shading, the performance does not depend much on resolution, because the graphics card's pipeline is busy with transformation of geometry.

Contrary, in case of expensive shading, the period increases less than twice when the resolution is doubled. While keeping same number of views, the frame period is linearly dependent on increasing the resolution.

In more complex geometric scenes, the performance directly depends on the number of views. In some non-optimized scenes, such as Asteroids, which dispatch many draw calls with trivial meshes, performance does not decrease linearly with number of views initially, because instancing of Geometry Shader can help in overcoming the overhead of many draw calls with simple geometry.

In conclusion, the exact effect on performance depends on **scene's complexity** and **complexity of shading**. For simple applications and lower resolutions, the effect can be minimal, in some cases negligible. Note that contemporary Looking Glass Display use 2500x1600 LCD display, which results in 4MPix. A 5x9 quilt with 512x512 texture size exceeds 11MPix.

8. Conclusion

The paper describes process and implementation of a conversion of arbitrary OpenGL 3D application with single-view output to multiview output by introducing a conversion layer and thoughtfully rewriting methods of OpenGL API.

The implemented software can be used as-it-is for bringing numerous applications such as legacy 3D games to autostereoscopic displays, and this has been verified for various applications using various rendering techniques. The impact on performance has been measured, and it strongly depends on the complexity of application's scene.

With minimal programming effort, the implemented injector could be extended to render stereo for Head-Up Displays, which may fill the hole of similar software for OpenGL as DirectX has been primarily aimed by such software in the past.

This paper has shown that it is possible to treat vertex transformation as a black box process and alter the transformation and projection at the same time. The convertor can be used to produce ground truth of multiview output, which could be used to aid further development of approximate single/stereo to multiview conversion methods.

In the future, the method and its implementation could be extended to support a broader subset of OpenGL applications, and to provide support for platform-conversion tools such as Wine[5], or to compensate for limitations such as frustum culling.

Acknowledgments

I would like to thank my supervisor Ing. Tomáš Milet for his help and knowledge. Finally, I can't forget to say thanks to my girlfriend, that still loves me despite spending evenings and nights mentally focused on my master's thesis.

References

[1] Nick Holliman. 3D Display Systems. 38, 12 2002.

[2] Quilts. https://docs.lookingglassfactory.com/KeyConcepts/quilts/.

[3] HoloPlay Core SDK. https://docs.lookingglassfactory.com/HoloPlayCore/HoloPlayCore-SDK/.

[4] BlueSkyDefender. Github: Blueskydefender/depth3d. https://github.com/BlueSkyDefender/Depth3D. Accessed: 2021-02-10.

[5] Features - vorpX - VR 3D-Driver for Oculus Rift, Apr 2018. https://www.vorpx.com/features/.

[6] Mostafa Mehrabi, Edward M. Peek, Burkhard C. Wuensche, and Christof Lutteroth. Making 3d work: A classification of visual depth cues, 3D Display Technologies and Their Applications. In *Proceedings of the Fourteenth Australasian User Interface Conference - Volume 139*, AUIC '13, page 91–100, AUS, 2013. Australian Computer Society, Inc.

[7] Alexandre Chapiro, Simon Heinzle, Tunç Ozan Aydın, Steven Poulakos, Matthias Zwicker, Aljosa Smolic, and Markus Gross. Optimizing Stereo-to-Multiview Conversion for Autostereoscopic displays. *Comput. Graph. Forum*, 33(2):63–72, May 2014.

[8] Petr Kellnhofer, Piotr Didyk, Szu-Po Wang, Pitchaya Sitthi-Amorn, William Freeman, Fredo Durand, and Wojciech Matusik. 3DTV at Home: Eulerian-Lagrangian Stereo-to-Multiview Conversion. *ACM Trans. Graph.*, 36(4), July 2017.

[5] https://www.winehq.org/