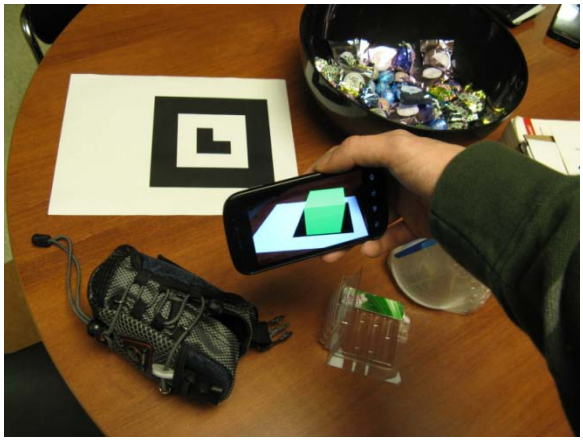
Advanced Rendering for Augmented Reality on Mobile Devices

Griffin Milsap
Eric Bourland

Abstract

Augmented reality on mobile devices was until recently only implemented with simple rendering techniques. We set out to add shader support to AndAR, the ARToolkit based Augmented Reality package for Android. In particular, we hoped to implement approximate environment mapped reflections and image space refractions.



Introduction

Augmented Reality (AR) is an amazing step forward in virtual reality technology. AR seamlessly meshes rendering of synthetic images into live video feeds through the aid of computer vision algorithms and rendering frameworks. An object can be rendered on top of a positional location indicator (often called a marker) which can relate important information about the camera's location relative to the marker through its projected representation on the video feed. AR is still a new technology, so the majority of research is focused on locating and tracking the marker with higher precision. Unfortunately, AR research doesn't often explore issues with the renderings themselves. Reflective and refractive objects which reflect and refract their environment have not been well explored and present some interesting technical issues that this project will explore. It is the focus of this paper to implement a method for approximating reflection and refraction in augmented reality renderings without any extra external data from the environment but the video feed.

In order to effectively perform reflective and refractive renderings in realtime, it is necessary to use a GPU and graphics library capable of fast shader support. Due to the nature of AR, it is preferable to render using the camera and screen of a mobile device due to portability and ease of use. As such, the Android platform was chosen for implementing this technology.

Android's ARToolkit based augmented reality package (AndAR [2]) is an existing software platform which runs on Android based mobile devices and is the subject of modification for this purpose.

AndAR does not currently support OpenGL ES 2.0, which is the updated version of OpenGL for embedded systems. ES 2.0 includes shader support, cube mapping, and other extensions to OpenGL. We set out to add some of the features of GLES2.0 to AndAR. In particular, we hoped to implement shaders for reflection and refraction, to fully take advantage of the graphics hardware in many mobile devices. For reflection, we used GL cube maps and environment mapped our objects. For refraction, we used an approximate image space refraction algorithm. AndAR is currently able to process the image fed to it from the device camera, find a specific marker placed in the scene, and calculate the transformation matrix required to render an object at the marker's location and orientation. We explore

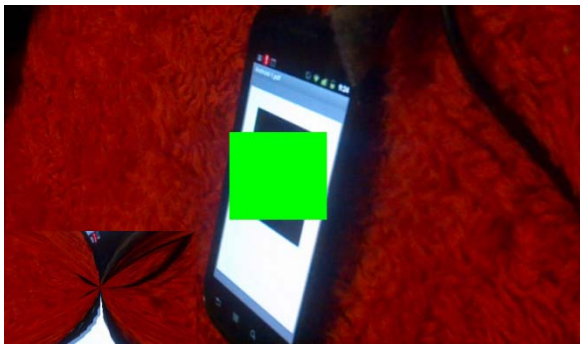
methods of adding additional interactive effects to the object rendered.

Reflection

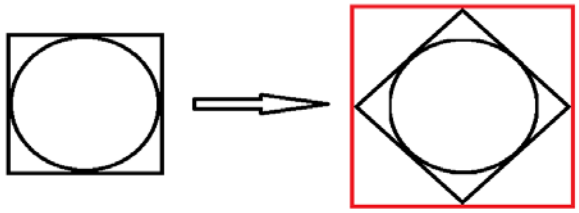
Rendering reflection in an augmented reality environment presents a very important problem. Information about the environment behind the camera is not available in the image and must be simulated effectively (read. faked). The process described in [1] by Ropinski et. al. is used to generate a cube map which can be used for rendering environment based effects. To render a reflective object, three operations must occur. First, the object's approximate screen space bounding box (SSBB) must be calculated. Next, the background image is segmented and rendered to a cube map texture. Finally, rays are fired from the camera per fragment to the object and are bounced at the cube map to find the appropriate fragment color.

Approximate Screen-Space Bounding Box

The algorithm described by Robinski et. al. assumes an accurate, axis-aligned object space bounding box (AABB). We apply the object's final transformation matrices to said AABB which entirely encloses the model to be rendered, then the screen space minima and maxima are calculated. This minimum (x,y) and maximum (x,y) constitutes the screen space bounding box (SSBB).



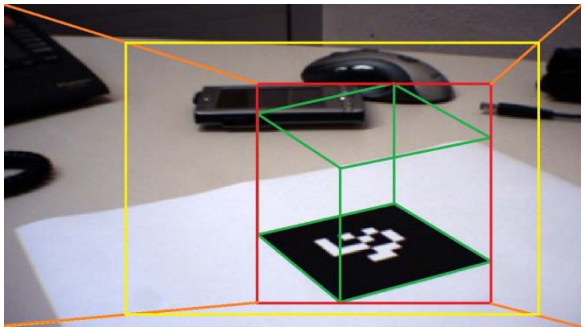
This algorithm doesn't necessarily return an exact bounding box, as depending on the shape of the actual object, the most conservative screen space bounding box may actually be smaller. For example, a sphere rotated 45 degrees will have a smaller SSBB than this method produces.



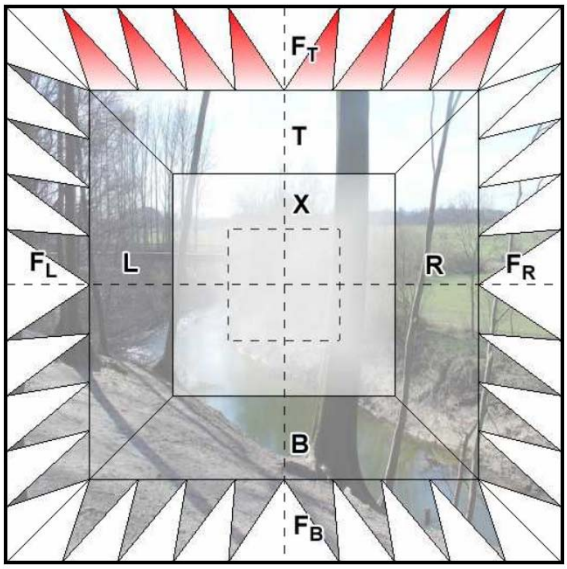
For our application, however, this approximate SSBB is adequate and yields massive performance enhancements.

Constructing the Cube Map

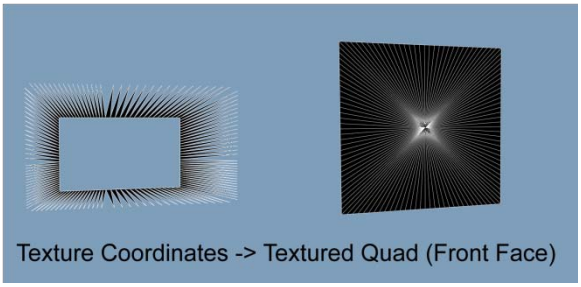
After finding the SSBB, we scale it up slightly and clamp it to certain screen boundaries as described in Ropinski's algorithm. We then find the midpoints of the lines from the corrected SSBB and the corners of the image.



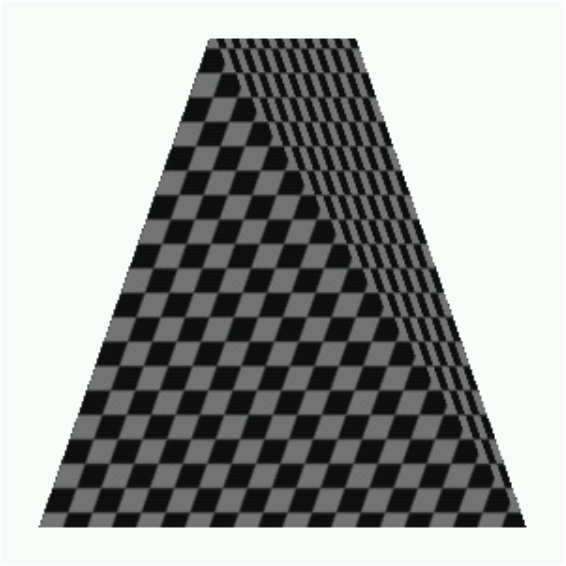
We then split the screen into the quads created by these lines and the two rectangles. The image contained by each of these quads is the image used for the sides, back, top and bottom of our cube map.



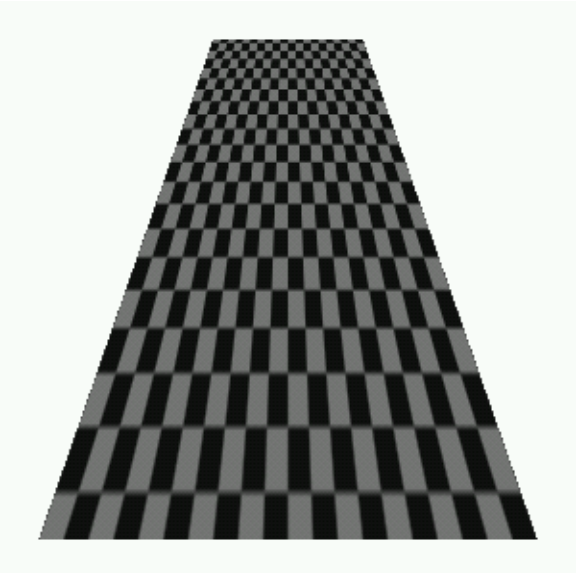
The outer remains of the image are used to fabricate an approximation of the front face. The texture coordinates are calculated for each of these faces individually, and geometry is generated which is then rendered directly into the cube map face texture via a framebuffer object. The front face consists of the outer border of the image cut into “n” triangles and arranged into one solid face.



This method essentially folds the cube map around the SSBB. Because GL triangulates all quads, the trapezoidal shapes created by this algorithm do not texture uniformly.



In order to correct this issue, the quads need to be assigned all four texture coordinates, instead of just the standard two. In this fashion, the texture can be treated as a projective texture and the artifact produced above can be removed. This method looks better, but is still incorrect in that the texture appears to be extending off into space but is actually all on the same plane. However, since the original texture is taken from an image that actually is extending off into space in most cases, our cube map should not be too adversely affected.



As it turns out, the actual textural artifacts caused by this non-uniform texture mapping is

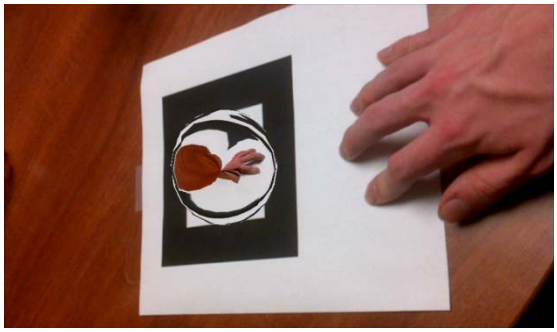
not visible unless under close scrutiny. These corrections should result in a negligible performance decrease, but the implementation complexity did not merit their inclusion in the final product.

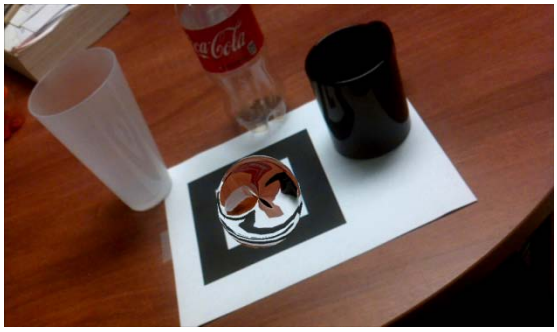
Reflecting in the Fragment Shader

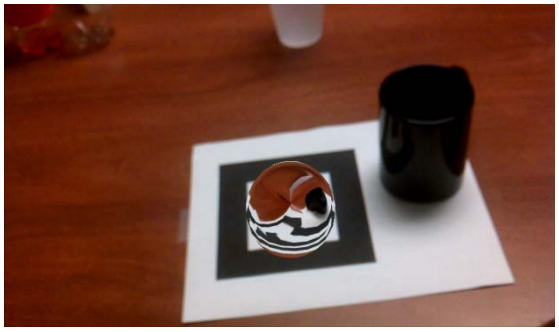
When the cube map has finally been constructed, we calculate the ray from the eye position to the object and reflect it about the normal interpolated across the face. GL returns the color at the point where the reflected ray intersects the cube map, and this color is assigned to the object at that fragment.

Results

The process produces this result at about 3 FPS on a Motorola Droid (1.0) phone, 12 FPS on the Nvidia Harmony Tegra 2 development kit (in debug mode), and 60 FPS on the Motorola Xoom tablet. The Motorola Xoom is the new Android 3.0 tablet which is built on top of the Nvidia Tegra 2 GPU/SOC.





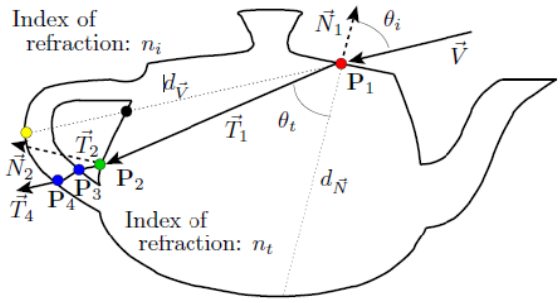


Refraction

We attempted to implement Wyman's algorithm for image-space refraction [3]. First, the ray from the camera to the object is calculated, and the approximate width of the object at that point is determined. Refraction occurs at both entry into and exit from the model which results in a very accurate result.

Approximating the Refracted distance

To find a rough estimate of the image's width inside of a shader, we render the image twice. The first pass, we reverse the depth test so farther faces are rendered and save the depth buffer to a texture. The second pass, we correct the depth test and save the depth buffer to another texture. Then, by subtracting these two texture values from each other we can find the approximate depth of the object at that point.



We did not further approximate this depth by using the distance along the normal to interpolate, although this yields more accurate results.

Refraction in the shader

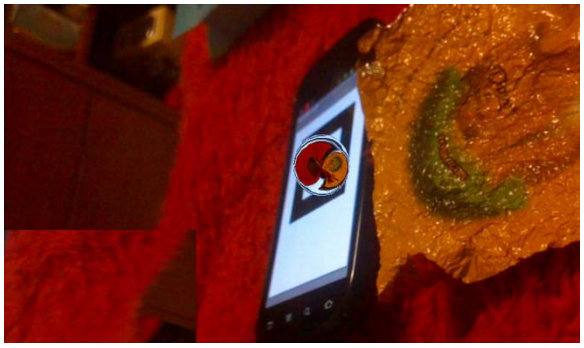
After constructing a cube map, we calculate the ray from the eye position to the object and refract it about the normal interpolated across the face using an index of refraction of 1.2. The refracted ray travels the distance calculated by our approximation, and then refracts back. GL returns the color where the ray intersects the cube map and this color is assigned to the object at that fragment.

Results

Our performance for this shader was similar to our reflection shader. However, due to the complexities involved with working on an embedded environment, our depth texture renders could not be configured properly, and would always fill entirely with solid black. There were several issues involved with depth textures that we explored, none of which ended up solving the problem. It is undefined behavior to write to a texture and read from it simultaneously, which we were doing inadvertently. It also causes severe performance issues to bind a framebuffer object (used to render depth textures) after doing any drawing to the screen on some devices, an issue we later corrected. Furthermore, many GL texture generation settings that are valid on PC implementations are not supported on embedded devices, and using them will crash the program. Due to time constraints, we could not get this multi pass method to work.

Conclusion

Despite the frustration caused by depth buffer rendering on a mobile device, reflection ended up producing moderately convincing results at interactive rates. Shader support and other updated features of GLES2.0 were successfully added to AndAR and they should be generally applicable to most meshes. Further work would hopefully get depth textures to render properly.



It took roughly a combined 100 hours to complete the project as it stands now. Griffin did the majority of the reflection and cube map code, and the updating of AndAR to GLES2.0 functionality. Eric did the first drafts of the shaders, and the refraction code, with Griffin tweaking and updating as necessary. Both group members spent most of their time debugging. This is unfortunately how embedded graphics development works.

Profiling tools such as Nvidia's PerfHUD ES which allow a look at the deep inner workings of the OpenGL implementation were invaluable in debugging these issues.

Project source code can be found on the andarshaders google code page.
http://code.google.com/p/andarshaders

A demo video can be found here.
http://www.youtube.com/watch?v=8cr5SLu02U0

References

[1] Ropinski, Timo and Wachenfeld, Steffen and Hinrichs "Virtual Reflections for Augmented Reality Environments", Proceedings of the 14th International Conference on Artificial Reality and Telexistence (ICAT04), 318(1), http://viscg.uni-muenster.de/publications/2004/RWH04/\

[2] Domhan, Tobias, Android Augmented Reality (AndAR), http://code.google.com/p/andar/

[3] Wyman, Chris, "An Approximate Image Space Approach for Interactive Refraction", ACM Transactions on Graphics 24(3), 10, http://www.cs.uiowa.edu/~cwyman/pubs.html

[4] Everitt, Cass, "Getting to know the Q texture coordinate...", http://www.xyzw.us/~cass/qcoord/