Magic Sprites - Sprite based refraction and other Sprite Effects

Matthew Fioravante *

Abstract

Modern 3d graphics hardware provides a host of capabilities for fast 2d sprite effects. This papers describes an implementation of several fragment shaders that give realistic details to simple sprites. In particular bump mapping, relief mapping, and sprite based refraction have been implemented. Performance and possible future direction is also discussed.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: sprites, surface effects, shaders

Links: DL PDF

1 Introduction

Modern 3d graphics applications such as games take an extraordinary amount of resources to create. While new technologies have enabled developers to create ever more realistic renderings, the costs and development time have also scaled at a geometric rate [vgs]. According to an online study, in 2010 the average development costs of a modern video game were around $28 million[Crossley].

Independent developers simply do not have the resources to develop 3d applications of this caliber. In addition, mobile platforms are becoming increasingly popular for 3d applications. Many of them support embedded 3d graphics APIs such as OpenGL ES and several low budget and indie video games have been developed for them.

The goal of this project is to investigate how we can leverage the features of the latest 3d hardware to produce 3d effects with minimal geometry. In particular the focus will be put on what effects can be done with sprites.

Sprites are easy to work with because they only require an artist to create 2d images. Details such as normals and depth can be added with additional textures. Unlike complex models, each sprite requires passing only a single vertex to the GPU, minimizing the bandwidth bottleneck on the PCI-express bus. When working with highly detailed sprites, we trade computation time on triangles and meshes for computation time in the fragment shader.

The approach will be to analyze the performance and trade-offs of 3 sprite shaders. Bump mapping, relief mapping, and refraction have been implemented using GLSL fragment shaders running in real time on the GPU. The aim of this work is to motivate further research and development into sprite effects.

*e-mail:fmatthew5876@gmail.com

2 Related Work

Many techniques have been proposed to produce realistic looking 3d effects without the need for extra geometry. Blinn et al. introduced traditional bump mapping [Blinn 1978]. Cook et al. [Cook 1984] introduced displacement mapping which is a technique to actually perturb the underlying geometry. This idea was refined by many others to produce parallax mapping [Kaneko et al. 2001; Tatarchuk 2006] and relief mapping [Oliveira et al. 2000; Policarpo et al. 2005; Policarpo and Oliveira 2006]. Both of these techniques add height data to flat geometry though use of a texture to produce self shadows, self occlusion, and parallax motion without any additional triangles. Another technique called impostors [Risser 2006] creates a 3d object using several height fields oriented at different positions. These can be viewed at several different angles. Layered depth information such as used in [Eisemann and Décoret 2006] can be used to produce volumetric effects and shadows.

All of these techniques can be used on sprites to produce realistic 3d effects with only 2d geometry. They can be used to augment an entire two dimensional environment such as a 2d game or to draw a three dimensional scene that contains sprites in place of models. In some cases, the use of camera facing sprites allows optimizations that were not possible before.

3 Implementation

The basic rendering framework was written in OpenGL using C++. It is written to be compliant with the OpenGL 4.1 specification using the core profile, and takes advantage of some of the newest API features.

Each sprite has 4 vertex attributes. The first is 3d position, the next is a integer frame number (for texture animations), a rotation angle, and finally a 2d scale factor in x and y. There are two classifications of sprites in the rendering engine, static and dynamic. For dynamic sprites, each of these vertex attributes can be updated every frame to do animations. Static sprites are created once and stored on the GPU for their entire lifetime.

In order to minimize data transfer to the GPU, each sprite is represented by a single vertex. The single vertex is transformed into a triangle strip to form a quad that always faces the camera in the geometry shader. Rotations are applied by a fast 2x2 matrix multiply. Scaling is simply a matter of scaling the initial positions of each newly created vertices out further from the origin. The frame attribute maps directly to the third texture coordinate of the texture array holding the sprite sheet.

The vertex shader does nothing more than pass through vertex attributes to the geometry shader. The fragment shader does all of the real work for each of the sprite effects described below.

3.1 Texture lookups and bump mapping

Two dimensional texture arrays were used to store the sprite textures. This allows for easy animation of sprites. Each layer of the array is a frame in the sprite sheet. Because a single texture array is used, we don't have to do multiple texture switches for each sprite

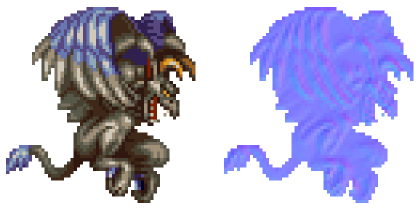

Figure 1: *Color map and Normal map*

sheet frame. An alternative approach would be to load the entire sprite sheet as a single 2d texture, and use texture coordinates to select each frame. That technique can cause bleeding artifacts at the edges of each frame from adjacent frames. Using texture arrays also allows one to control the clamping or repeating behavior of the texture at the edge of the sprite.

With regards to normal vectors of screen facing sprites, tangent space coordinates are the same as eye space coordinates. Therefore we can save some computation in places where traditional algorithms for 3d geometry would need to compute transformation matrices to go from one space to another. In particular bump mapping becomes nothing more than one extra texture lookup. Figure 1 shows an example color map and normal map that will be used in the performance benchmarking results in section 4.

3.2 Relief Mapping

Relief mapping is a technique that is used to create the illusion of height based detail on a flat surface. For each fragment, a ray is cast from the eye position into the depth texture to find where the ray intersects the underlying surface. First linear search is used to quickly find the general area where the intersection is located. Then binary search is used to get closer to the exact point of intersection. The number of linear search steps must be large enough to not be able to skip over thin features. Adjusting the number of binary search steps improves accuracy and reduces aliasing at the cost of performance. The technique was originally proposed by Oliveira et. al. [Oliveira et al. 2000] where they used image warping techniques and software based rendering. Later Policarpo et. al. [Policarpo et al. 2005] implemented relief mapping in real time using fragment shaders.

Relief Mapping is implemented in the fragment shader using the implementation provided by Policarpo et. al. [Policarpo et al. 2005] as a base. Relief mapping works in tangent space, but since we are using sprites we can work directly in eye space and avoid computing transformations. This implementation supports self shadowing, fragment depth, and silhouettes. It was also extended to support multiple light sources. Each active light source requires an additional pass through the relief mapping algorithm to compute shadows.

3.3 Refraction

The most notable contribution of this work is sprite based refraction. Sprite based refraction is based on relief mapping. This time, 2 depth textures are used to define a volume of a refractive object. Figure 2 gives an illustration.

The actual geometry of the sprite (quad) sits on the topmost layer which is represented by the purple line in the diagram. The thicker blue line is the color texture of the sprite. Let R be the refractive sprite we wish to render.

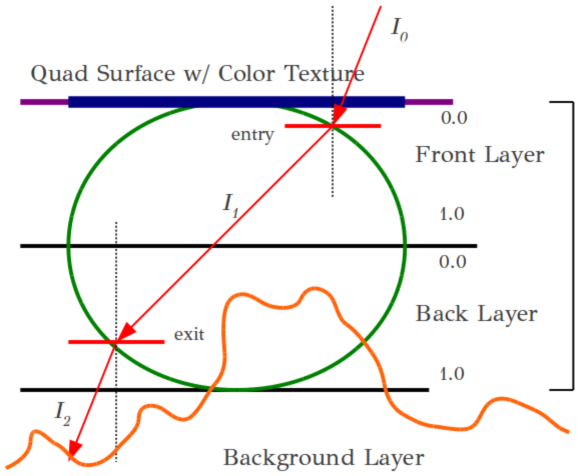

Figure 2: *Refraction diagram*

First, render everything in the scene other than R. We take the color and depth values in the frame buffer and store them in textures. Next we load these textures and render our sprite R.

As seen in the diagram, I_0 is the incident vector from the eye to the fragment on the surface of R. First, we must find the entry point on the surface using the traditional relief mapping algorithm on the top level depth texture. Once we find the entry point, we can lookup the surface color and normal at that position. A quick alpha test is done here, and if the surface color value alpha is 0, we can discard the fragment immediately. The fragment depth is updated and depth testing is performed to check for occlusion. If the fragment is occluded we can also throw it away without needing to continue processing.

The standard lighting computations are also done here at the relief surface. This implementation uses blinn-phong but any other lighting model could be substituted. Since the object is translucent, shadow computation was disabled as an optimization. Multiple lights are supported and the results of each lighting computation at the surface are added up and saved until the blending step at the end.

Given a ratio of indicies of refraction η, the next step is refract the incident vector I_0 into the surface to get a new incident vector I_1. The standard **refract()** function supplied by the GLSL API is used here.

Now that we have the incident vector I_1, we do another relief mapping pass on the bottom level depth texture to determine the exit point. In addition, we also need to check whether I_1 collides with any of the background geometry. We do a third relief mapping pass with I_1 on the frame buffer depth texture, which is represented by the orange line in figure 2.

If I_1 collides with the background we are done and can move onto the blending step at the end. If not then first we invert the ratio of indices of refraction $1/\eta$ and refract I_1 out of the surface the exit point to get I_2. Finally, we do the last relief mapping pass on the frame buffer depth texture and I_2, starting from the exit point, to determine where the ray hits the background.

Now that we have a surface color value and and a color value of where the incident ray hits the background, we blend the colors together to get the final result.

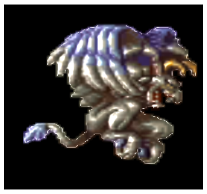

Figure 3: *Bump mapping example*

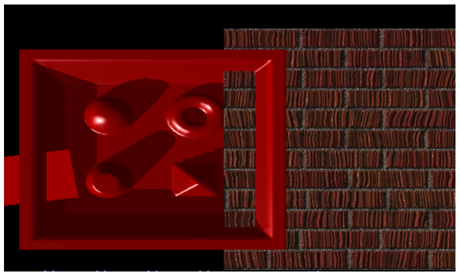

Figure 4: *Relief mapping example*

4 Results

Figure 3 shows an image of a gargoyle with a normal map applied. Diffuse and specular lighting can be seen. Compare to the original image in figure 1.

Figure 4 shows an example of relief mapping with self shadows and fragment depth. The shadows on the red object are produced in real time by the relief mapping algorithm. A light source moving across the scene generates moving shadows. The bricks also use relief mapping to produce shadows and diffuse lighting. There are three objects in this scene and each one is a single sprite. Fragment depth testing can also be observed where the brick sprite and the red relief sprite intersect. The normals in the relief texture produce the diffuse lighting and specular highlights.

Figure 5 shows an example of a spherical glass ball doing refraction in real time. In the demo application the ball can be moved over the scene. Objects inside of the ball appear differently than objects behind it. Diffuse and specular lighting can also be seen on the surface. Since this algorithm only uses the depth buffer to determine collisions, it can be used with any kind of rendered or pre-rendered scene.

The system used to render these scenes was a Core i7 860 at 2.80GHz with 16GB of memory running Ubuntu Linux 10.10 using the binary NVIDIA driver version 209.19.06. The GPU is an NVIDIA GTX 470 with 1280 MB of GDDR5 memory. Due to lack of time, neither the shaders nor the client side code have been profiled or optimized.

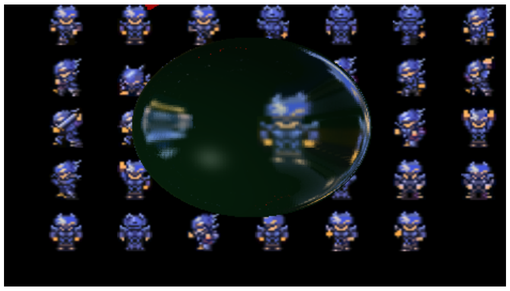

Figure 5: *Sprite based refraction example*

Table 1: *Performance benchmarks for unoptimized code*

Scene Data	1080p FPS	800x600 FPS
Bump mapped animated sprite	5000	10000
Relief mapped red box	700	2500
Relief mapped brick sprite	357	1428
Refraction Sprite with background	400	1500
All of the above	210	588

Table 1 lists some benchmarks with different scenes. Each scene had 2 directional lights. One light was fixed pointing in positive X, Y, and Z. The second light was rotating around the Y axis at a fixed speed. Benchmarks were taken at a full screen resolution of 1080p (1920x1080) and a windowed resolution of 800x600.

The results turned out pretty good for unoptimized code. As expected, screen resolution has a significant impact on performance. All of the work is being done in the fragment shader. When rendering the relief mapped sprites, the performance would immediately improve if the camera was moved so that part of the relief sprite was occluded.

The test system is a pretty high end system. It would be desirable to increase performance to allow rendering of many sprites with complicated shaders and also use on older systems that aren't quite as powerful. While the rendering engine was written to OpenGL 4.1 core, it could be back ported to support older versions of OpenGL.

5 Limitations and Future Work

The relief mapping algorithm given by Policarpo et. al. can produce aliasing. In particular, the shadow produced by the pyramid object in figure 4 exhibits some aliasing. Higher resolution relief maps can be used to reduce the aliasing but not remove it entirely. Increasing the number of binary steps can also reduce aliasing, at the cost of performance. Relaxed cone stepping [Policarpo and Oliveira 2007] could be used in place of the binary search approach used by traditional relief mapping to alleviate these aliasing problems. Relaxed cone stepping would also improve performance at the cost of one-time pre computation. As another optimization, fixed shadows and highlights could be baked in to the color textures.

The refraction based sprite algorithm is still widely untested and has not been thoroughly analyzed. Since it depends on the frame buffer being rendered to a texture, it will not work correctly if the refracted rays shoot off outside of the frame buffer region. The refracted sprite must not get too close to the edge of the screen. Depending on the scene, this problem could be avoided by repeating the background texture, choosing a fixed color, or using some other texture lookup for rays that make it outside of the frame buffer region. At the cost of additional memory and computation time, a larger window could be rendered into the background texture. One could also do a separate rendering centered at the sprite to render refractive sprites on the edge of the screen.

It is not clear what classes of shapes and ratios of refraction (η) are acceptable. Certain combinations may produce rays that shoot off too far past the background. For the initial implementation a simple sphere relief map was generated. Other simple shapes should also be possible. For example one could render a glass of water. If one only desires to render a basic shape such as a sphere, a procedural approach would be much faster. Intersection points could be computed directly and texture memory and lookups for relief maps could be avoided entirely.

The refraction algorithm requires up to 4 relief mapping passes. The shader code should be analyzed and profiled carefully for optimizations.

6 Conclusion

An implementation of bump mapping, relief mapping, and refraction based sprites has been presented and analyzed. These techniques illustrate how sprites can be used in place of complex models in three dimensional scenes to add realism. The performance, quality, and work flow trade-offs between these mapping techniques and creating complex models must be considered before choosing which implementation to use. This work is only the beginning of the possibilities of what can be done with sprites using 3d graphics hardware.

References

BLINN, J. F. 1978. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph. 12* (August), 286–292.

COOK, R. L. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '84, 223–231.

CROSSLEY, R. Average dev costs as high as $28m - game development - news by develop. http://www.develop-online.net/news/33625/Study-Average-dev-cost-as-high-as-28m.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '06, 71–78.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., AND TACHI, S. 2001. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, 205–208.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 359–368.

POLICARPO, F., AND OLIVEIRA, M. M. 2006. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '06, 55–62.

POLICARPO, F., AND OLIVEIRA, M. M. 2007. *GPU Gems 3 Ch. 18 Relaxed Cone Stepping for Relief Mapping*. Addison-Wesley Professional.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. A. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '05, 155–162.

RISSER, E. 2006. True imposters. In *ACM SIGGRAPH 2006 Research posters*, ACM, New York, NY, USA, SIGGRAPH '06.

TATARCHUK, N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '06, 63–69.

Video game costs - video game sales wiki. http://vgsales.wikia.com/wiki/Video_game_costs.