FrequencyShift

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Oliver Kuckertz * Timothy Blut † Kevin Diehl ‡ Karl Mertens § Arthur Drichel ¶ Malte Modlich ‖

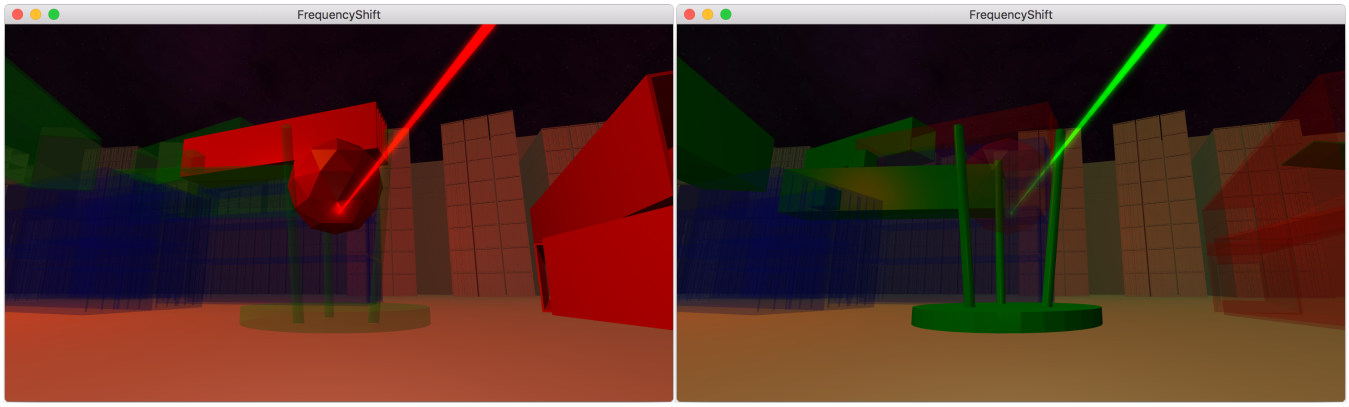


Figure 1: *The teaser image is basically an eye catching image making your report interesting at first glance.*

Abstract

FrequencyShift is a cross-platform, multi-player shooter game developed for the practical course "Dive into Mobile VR/AR Games". The basic game idea was to develop a virtual version of the game laser tag. In order to make the game graphically challenging and require more tactic during gameplay in comparison to common shooter games, game world objects and players are each assigned a frequency, encoded as color, which influences transparency and collision testing. For example, figure 1 shows the same world from three different frequencies. Note how the laser collides only with objects that match its color. In this paper, the development process, software structure and relevant implementation details of FrequencyShift are explained.

Keywords: game programming, first-person shooter, transparency, forward shading

1 Gameplay

Prior to introducing the high-level gameplay mechanics, the reader must understand the game's concept of frequencies. Let it be said that this game is not physically accurate - this is not how stuff really works; Star Gate showed us that one must shift in phase, not frequency, in order to pass through other objects - and should not be cited by any serious physics or optics paper. That being said, let's continue:

FrequencyShift adopts common game mechanics of first-person shooter games. The player can control his position, camera and body frequency in the world using various input devices. Support for mouse, gamepad, keyboard and orientation sensor input is implemented.

*oliver.kuckertz@rwth-aachen.de
†timothy.blut@rwth-aachen.de
‡kevin.diehl@rwth-aachen.de
§karl.mertens@rwth-aachen.de
¶arthur.drichel@rwth-aachen.de
‖malte.modlich@rwth-aachen.de

There are three different frequency modes: None, 3 or 5 frequencies. When the mode is set to none, the render engine and game model will act like any common first-person shooter. Each mode implements a distance function between two frequencies: The distance is the absolute difference of both frequencies' index values over the field of available indices. Therefore, when no frequencies are available, the distance is alway zero; with three frequencies, the maximum distance is 1, and with five frequencies, the maximum distance is 2.

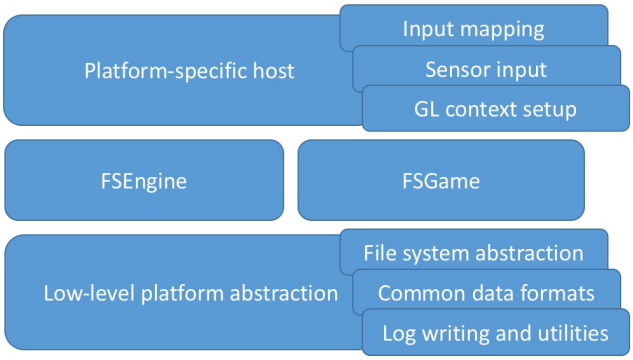
In mode 3, the available frequencies are red, green and blue. Mode 5 adds yellow between red and green, and purple between green and blue. (This again is not physically correct because blue and purple are swapped, but allows for easier calculations within the field of available frequencies across all modes.) An additional special "white" frequency is defined with a distance of 0 to all other frequencies.

When performing collision checks, the game model only considers objects which have a frequency distance of zero to the colliding object. However, further optimizations described in section 5 have been applied for collision checking. The distance function is only used for deriving transparency values.

2 Development process

Development began by creating a proof-of-concept OpenGL application that could be run under both iOS and OS X with minimal differences in code. Patching up differences was done using the C preprocessor. Next, basic implementations of the components FSEngine and FSGame were added. A message-passing interface was created to let the components communicate through pointers that had been shared through the host process. From then on, development of FSEngine and FSGame progressed independently for several months. Once game model and engine were completed, a graphical interpretation of the world was developed and added to FSEngine.

3 Platform abstraction



In our group, only two out of six people had permanent access to Apple desktop hardware, and only one out of six had permanent access to an iPhone 6. In order to still function as a team, we decided on implementing abstraction layers which encapsule platform-specific behavior and setup routines. IDE project files and makefiles for all supported platforms were created and maintained independently.

4 Rendering pipeline

4.1 Main rendering

Transparency is a key gameplay concept; in order to correctly apply effects to transparent objects, the renderer uses forward shading. Support for opaque and transparent objects is implemented separately; grouping is done using a render bucket concept. A frustum culling phase filters all invisible objects when rendering from buckets.

The rendering pipeline operates in 4 seperate passes: First, all opaque objects are rendered and the depthbuffer is filled. For all following passes, writing to the depthbuffer is disabled, but reading is enabled so that pixels that would be hidden by an opaque object are not processed by the fragment shaders. A skybox is rendered during the second pass. In the third pass, all objects which require additive blending (such as particle systems and laser beams) are rendered. Particles are batched into a single buffer and rendered with instanced billboards. In the fourth pass, all the transparent objects are rendered with alpha blending. The objects are sorted from back to front so that objects in the back of the scene can be seen through closer objects.

4.2 Post processing

Rendering of laser beams works by rendering a beam mesh onto a low resolution texture, which is then blurred with an optimized gaussian blur shader described under section 4.3. The blurred image is rendered onto the final image along with additional post-processing effects, including screen tinitng and a vignette texture. Rendering the blurred images onto the lasers achieves a glowing look.

The screen tinting effect is used for tinting the final image in the current frequency color. Additionally, the effect is applied to slowly fade the scene to black when the player dies. The vignette texture is used as a red colored hit inidicator.

4.3 Optimizations

Blurring is improved by stretching a low resolution using linear sampling. This approach saves additional bandwidth [Int July, 2014]. The renderer batches up to 4 lights into a single draw call to reduce the amount of overdraw. This greatly improves the performance on devices with low fillrate. Furthermore point lights are also culled based on camera frustum so no lights are computed which would be invisible from the camera perspective.

In order to avoid synchronization between the CPU and GPU, all lights and object transformations are uploaded into large buffers before starting the rendering process. Many matrix computations are avoided due to the engine's event based design. When an event affects a specific transformation, it is recomputed on demand.

5 Physics

The Bullet Physics library [Bul June, 2015] is used for collision detection and ray tracing in FSGame. Engine and host components do not interact with Bullet Physics, but merely receive messages generated by FSGame due to physics events.

Initial attempts at integrating Bullet Physics with our frequency concept failed due to incompatibilities with Bullet's built-in collision group and mask system. Either expensive (due to cache misses) function calls would have to be made in a Broadphase callback, or redundant objects would have to be created to fit into Bullet's group system. Neither option was feasable.

The final solution which resulted in improved performance when compared to frequencies being entirely absent was to emulate the frequency distance function using the bitwise and operator. All data required for this simple computation step could be stored in a Bullet collision proxy object and therefore the CPU's internal caches.

6 Map preprocessor

A map compiler was written in order to compensate for long load times. Originally, loading the map "Container" with an unoptimized build of FrequencyShip took up to 40 seconds on the iPhone 6. The load times could be greatly reduced by preprocessing the map: The Wavefront Object file is parsed by the compiler, re-encoded to a format used by the game model, and optionally preprocessed by Bullet Physics. The resulting binary file could be loaded into the memory using the "mmap" syscall, which results in asynchronous on-demand loading of the map through the operating system kernel. After applying these optimizations, map load times have been reduced to less than two seconds.

7 References

References

June, 2015. Bullet physics library. http://bulletphysics.org/.

July, 2014. An investigation of fast real-time gpu-based image blur algorithms. https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms.