

Realtime Graphics VU Documentation

Timon Höbert, 01427936, Stefan Sietzen 00372194

General structure

The whole system is implemented in C# using the OpenTK C# bindings of OpenGL. The system is separated into the particle system and the polygon system.

Asset loading

The geometries are loaded using the Open Asset Import Library (Assimp.NET). Hereby the faces, vertices, normals, UV-coordinates, and material lighting properties are loaded from Wavefront .obj and .mtl material files. The material files reference the used texture, which will be loaded as well in the PolygonObject class. The loaded data is stored in the particular OpenGL-buffers or as a parameter for the shader program.

Particle System

The particle system is implemented with two textures each for positional and velocity data, which act as a double buffer, as well as 3 nearest neighbor textures containing information about accumulated positions + velocity in voxels of the total scene space and a count to be able to calculate the averages. The base position/velocity textures are rendered by transforming one point per particle with the corresponding values to a unique, vertex-id-based 2d position corresponding to one pixel each, the nearest neighbor texture is rendered by mapping a 3d voxel position to a 2d texture position and rendering the particles to the pixel corresponding to their current voxel coordinate (with additive blending). The nearest neighbor textures are then queried in the physics shader to get influence by surrounding particles according to the "boids" algorithm. This is obviously only an approximation suffering from artifacts but leads to a much-improved performance (linear runtime) over an accurate boids algorithm (quadratic runtime when implemented naively) enabling real-time simulation of 4 Mio. particles on a 1070 GTX graphics card.

Collision handling is done for the walls and the spheres in the scene by approximating an intersection point with the average of the old and the new position (upon a detected penetration) and reflecting the velocity, scaled by a dampening value, based on the surface normal.

For the first 2 seconds, a vectorfield affects the particles. This vector field has been simulated in an external 3d software (Houdini) and exported via python script to an ASCII based format.

Shading

The lighting on polygons is done using the Phong reflection model for ambient, diffuse and specular light/highlights. This is calculated using the position of the camera and the light source and the normal of the shaded polygon. Particles are colored based on vertex ID and additively blended.

High Dynamic Range and Tone Mapping

The whole scene containing the polygons and the particles are shaded into a framebuffer for offscreen effects, in this case, High Dynamic Range (HDR). This functionality is encapsulated into the OffscreenRenderer-class. The framebuffer consists of a float32-texture for the RGB-color values and a renderbuffer for depth values. This framebuffer is activated before rendering the particles and polygons. The resulting texture with a higher intensity range is used as input for the hdr-shader which applies Reinhard tone mapping. The final post-processed output is rendered as a flat texture on the screen.

This HDR implementation avoids the large blown out areas where the particles become dense because of the additive blending of the particles.

Audio Processing

The background music track is played using the NAudio library. The mp3-file is live processed to change the scene accordingly to the music. A discrete Fast-Fourier-Transformation is applied to calculate the music intensities for the whole frequency spectrum.

Point Light Shadows

The point light shadow system uses cube maps and a geometry shader technique for rendering the required depth information in all 6 directions. Our implementation is basically a port of this tutorial: <https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows>

Light Maps

The final Scene has been modeled in the 3d Software Houdini, where we also rendered 5 different lightmaps with global illumination from distinct groups of lights. Originally we wanted to animate the spheres in the scene according to the audio spectrum, but we switched to animated lights instead. The lightmaps were generated with the GPU Renderer "Redshift". Despite being calculated on a powerful GTX 1070 GPU, this process takes an entire night of rendering (we rendered 4K maps which we then downsampled to 3K). Also, to be able to add the lightmaps together, they have to be exported (and imported) as floating point images in linear space. We used a 16-bit float TIFF format for this, as it was the only float format that the .NET Image Class supports, also using 16-bit instead of 32 saves a lot of space. 4 of the 5 light maps (the ones of the circularly-arranged area lights) are influenced by the audio spectrum while one is statically displayed as room ambiance.

On the ceiling there is was an error with the lightmap generation where some UVs didn't get unwrapped correctly, so to avoid the time-consuming generation of new lightmaps, we simply covered the area by a larger circular light (originally there was a smaller one in the same area).

Please be aware that the initial loading of the lightmap textures can take a while (~10 sec. on our SSD-equipped system).

Test System

Unfortunately, we were not able to test the System on the Lab PC, but our own testing systems had NVIDIA cards installed.

The song takes about 4 minutes, there is not really any change except the procedurally animated lights, so you can stop whenever you think you got enough of an impression of the effects. Otherwise the application exits after the song.

Controls

The camera movement mode can be switched with the V-key on the keyboard. The manual camera can be controlled using the WASDQE-keys. If the camera is in automatic mode, these keys also switch into manual mode. The whole demo can be quitted using the ESC-key.