Particularly Shady

Alexander Cech (08900070)
Realtime Rendering 2019W

Final Submission Report

The original project proposal and updated references can be found in Appendix A.

Work for this demo has focused on implementing the proposed effects, namely a particle system, GPU-sorting, opacity shadow mapping, conventional shadow mapping, curl-noise, and bringing it all together with solid geometry in a playful scenery.
Some ideas (but none of the required effects) from the initial proposal have been scratched (particle emission/attraction to/from textures); instead bone animation for models, which was not in the propsal, was implemented.

About running the demo:
Fullscreen mode and size can be configured in the file ***assets/config.ini*** .
By default, the demo switches to full screen mode with the current desktop resolution. Development and tests where conducted with a resolution of 1920 x 1080; sufficient frame rates for higher resolutions can not be guaranteed.
The demo terminates after the ending. It can also be interrupted prematurely by pressing the ‘Esc’ key.

Debug controls:
The demo features an automatic camera; a debug menu can be activated by pressing the ‘G’ key. This was used for debugging and experimenting with parameters, as well as to modify geometry and light setup.
‘F1’ toggles wireframe mode, ‘F2’ toggles backface culling, ‘F’ toggles fullscreen mode.
‘Esc’ terminates the demo.

Scene description:
Textured and untextured geometry is rendered, along with a particle swarm that is affected by curl noise. The particles are rendered with proper alpha-blending; for this they are sorted on the GPU before rendering. The scene is illuminated by a (moving) directional light source (“sun”); shadows of the semi-transparent particles are calculated and cast inside the particle swarm itself, as well as onto solid geometry. Shadows from solid geometry are received by the particles as well.
The demo begins with a thin particle smoke simulation; particles are affected by potential fields, using curl-noise for semi-realistic air swirls and turbulence.
In the middle (“monkey”)-part of the demo, additional gravitational simulation and bouncing of particles off the scene surface is activated.
In the following part (yellow “dumbbell”) the combination of semi-transparent particles, their self-shadows and shadows from geometry can be seen nicely.
The next part (“tornado”) features particle traces to increase the flow perception.
Model animation occurs during all parts of the demo, using two base animation patterns (one for the goblins, one for the dragon); the animations for each goblin are slightly offset from each other in order to avoid a “lockstep” look.

Vulkan framework:
A basic framework to deal with Vulkan initialization and the rendering loop has been developed, as well as a bunch of wrapper functions to handle repeating low-level function calls. The framework is substantially inspired by [Wil15] and [Kap18].

Particle system:
All particle calculations and data stay on the GPU; only a uniform buffer with control parameters is transferred each frame. 2 variants of the system have been experimented with: Variant A ("indexed scheme") uses a single particle data buffer, along with double-buffered "alive-lists" that index into the data buffer. Variant B ("non-indexed scheme") uses a double-buffered particle buffer and no index-lists (meaning that the decision whether to draw a particle or not has to be made by the rendering shader). Despite that overhead, variant B is more performant due to one less level of indexing and better cache coherency, and is the one used in the final demo.
Particle simulation is currently split into two main parts: A physics-based part deals with force-based attraction/repulsion (e.g., for gravity and plane-collision), and a velocity-field-based part deals with emulating curl noise. Both approaches can be combined as well.

Opacity Shadow Maps:
OSMs are generated each frame, by doing an orthographic rendering of the particles with the view direction of the light source. Preliminary "auto-focusing" is implemented, to calculate suitable values for the projection matrix so that the particle swarm is closely encompassed. For each particle, its distance from the near plane (light source) is used to assign it to one of the 4 color-channels of 4 render-targets, where the particle alpha-values are accumulated, resulting in a 16-layer-opacity map.
When rendering (particles or solid geometry), the fragment position is transformed to shadow-map-space; an accumulating lookup is done through all layers closer to the light source than the fragment. The final light transmittance obtained by this sum is used to calculate the amount of shadowing.

Conventional Shadow Maps:
Not much to say about those – Percentage Closer Filtering (PCF) is used to obtain smoother shadow borders. Slope-dependent biasing is used to avoid "shadow acne" and "Peter-Paning".

Curl-noise texture:
Calculating multiple instances of time-varying perlin or simplex noise per particle (for emulating curl noise) directly in the particle compute shader simulation was not feasible for performance reasons. Instead, the potential field from which velocities are derived is calculated on the CPU asynchronously: The RGB channels of a texture represent the XYZ components of a potential-vector. Three CPU threads calculate time-dependent perlin (or simplex) noise, each thread responsible for one of the texture channels. Once a new texture is available (typically every couple of frames), it is uploaded to the GPU and used in simulation. (The potential field was set up so, that each potential component depends only on the other two components of the particle position, thereby avoiding the need for a 3D-texture).

GPU-Sorting:
Two sorting methods are implemented: Radix-Sort and Odd-Even-Merge-Sort (and for each of those, a separate implementation for the indexed and non-indexed particle system scheme). While Radix-Sort is faster, it still limits the number of particles, as a full sort is done every frame. Odd-Even-Merge-Sort has the property, that it can be split into independent rounds, where each round leaves the data in a better sorted state than before. This allows to not do a full sort each frame, but only a couple of rounds, where particles get progressively sorted better and better. Since there is strong frame-coherency with regard to the depth-value of particles (no sudden camera jumps), this is good enough to not cause visual artifacts.
One problem that does arise with progressive sorting, is that newly emitted particles are usually far away from their final sorted position in the buffer, which **is** noticable. To counter this undesired effect, newly emitted particles start in a “delayed” state, during which they are not rendered, but still progressively sorted. Only after a short time they are then switched to “active”.
OEM-Sorting and the delayed-state-trick were inspired by [Syl07].
The final demo uses the progressive OEM sort method.

Geometry rendering:
All solid-geometry is rendered in one single draw of *vkCmdDrawIndexedIndirect*. For this, the vertex and index buffers of all used models are combined into one big buffer respectively, and textures are passed in the Vulkan-analogon of bindless mapping, i.e. by using an unbound sampler array. Model-matrices, colour and texture indices are stored in a GPU-resident buffer, that is updated only upon changes. Along with a buffer of draw references (one per model type, containing the required number of instances and offsets) all information to render all models in one go is complete.

Bone animations:
These have been implemented in addition to the original proposal.
Bones are read from the model file, and interpolated in time. The current bones transforms and weights are used in the geometry-render shader to obtain the final vertex positions.

Libraries:

Assimp	http://www.assimp.org
OpenGL Image	http://gli.g-truc.net/0.8.2/index.html
GLFW	https://www.glfw.org
Dear ImGui	https://github.com/ocornut/imgui
GLM	https://glm.g-truc.net/0.9.9/index.html
mIni	https://github.com/pulzed/mINI
FMOD (core)	https://www.fmod.com

Resources:

Goblin model	from Sascha Willems’ Vulkan Examples [Wil15]
Dragon model	by “3dhaupt”, https://free3d.com
Music	YouTube Audio Library, https://www.youtube.com/audiolibrary

GFX card used for testing and development:
NVIDIA GeForce GTX 1050 Ti

Appendix A: Original proposal

The main focus of this demo lies on realistically rendering shaded particle systems, to simulate volumetric media like thin smoke or dust.

In particular, semi-transparent particles will be lit by a (non-static) directional light source, taking into account the varying light transmittance through the particle volume. Thus, particles will be affected by self-shadowing, and cast denseness-dependent shadows to solid geometry in addition. Vice-versa, solid geometry will also cast shadows on particles.

In order to achieve a realistic look, the particles must be rendered in back-to-front order with proper alpha-blending (**not** additive blending), which means that it is necessary to sort them before drawing.

Emission of particles will be implemented via different methods: Besides classical point emitters, emission from geometric primitive volumes and/or surfaces, as well as emission from mesh surfaces and from textures is planned.

In a similar fashion, attraction towards textures is planned, e.g., to form text, logos or images from an initial chaotic dust cloud over time.

To increase realism, especially for smoke representations, (fake) turbulent flow will be simulated by curl noise.

The scene layout / choreography for the demo is not yet completely determined; most likely the background scene will be an enclosed space or “arena”, in which various particle effects will be showcased in a playful way.

The main challenge will be to be able to handle a sufficient number of particles to achieve a realistically looking representation of semi-transparent media in realtime.

As API, Vulkan 1.1 will be used.

(Effect list and references on next page)

Main features/effects:

Opacity Shadow Maps [Kim01]
To implement particle self-shadowing and shadow-casting to solid geometry, Opacity Shadow Maps will be implemented.

Conventional “plain” Shadow Maps
For casting shadows *from* solid geometry, conventional shadow maps will be used.

Curl-Noise for particle flow [Bri07]
To simulate / fake turbulence in the represented medium (smoke, dust).

Own extensions to curl-noise
Extensions to modify a velocity field, based on the math of the Curl-Noise paper, are planned, to achieve visually pleasing effects, like wind and tornados.

GPU sorting [Har11, Kip05, Lin09]
Sorting must be done on the GPU to allow for reasonable numbers of particles, so an appropriate sorting algorithm will be implemented. (Probably radix-sort or odd/even merge sort).

Emission from points, surfaces and/or volumes
including emission from mesh surfaces

References

[Bri07] Robert Bridson, Jim Houriham, and Marcus Nordenstam. 2007. Curl-noise for procedural fluid flow. In *ACM SIGGRAPH 2007 papers* (SIGGRAPH '07). ACM, New York, NY, USA, Article 46.

[Har11] Takahiro Harada and Lee Howes. 2011. Introduction to GPU Radix Sort. In *Heterogeneous Computing with OpenCL*.

[Kap18] Arseny Kapoulkine. 2018. Building a Vulkan renderer from scratch. https://www.youtube.com/playlist?list=PL0JVLUVCkk-l7CWCn3-cdftR0oajugYvd

[Kim01] Tae-Yong Kim and Ulrich Neumann. 2001. Opacity shadow maps. In *Proceedings of the 12th Eurographics conference on Rendering* (EGWR'01). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 177-182.

[Kip05] Peter Kipfer and Rüdiger Westermann. 2005. Improved GPU Sorting. In *GPU Gems 2*. https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html

[Lin09] Linh Ha, Jens Krüger and Claudio Silva. 2009. Fast 4-way parallel radix sorting on GPUs. In *Comput. Graph. Forum. 28*. 2368-2378.

[Syl07] Sebastian Sylvan. 2007. Particle System Simulation and Rendering on the Xbox 360 GPU. *Master Thesis Project in Computer Graphics, Chalmers University of Technology.*

[Wil15] Sacha Willems. 2015. Vulkan examples. https://www.saschawillems.de/creations/vulkan-examples/