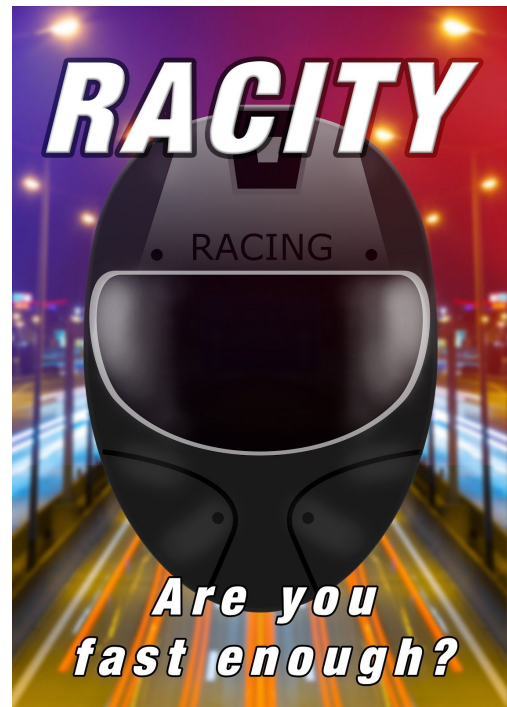


RaCity

Features

- Complex textured models
- NVIDIA PhysX Engine
- NVIDIA PhysX simulated vehicle
- Real-time smoothing of different objects using Subdivision Surfaces on GPU
- Fully GPU-animated Particle System
- Procedural smoke and fire textures
- HUD with debug information and leaderboard
- Highscore persistence with SQLite
- Lightmapping for street texture to display shadows
- View frustum culling with Bounding Spheres



Story

A son, always racing with his father's car through the city and damaging it caused his father to take drastic measures and mounting a contact sensitive bomb into his car. Will his son still be able to race through the city without getting any scratches and bumps into the car?

Freely movable Camera

By default the camera follows the player's vehicle and can be detached (by pressing the C-Key) to freely mode for debug viewing. The vehicle's follow camera is implemented by inverting the offsetted model matrix of the vehicle. The freely camera is a full third person camera and can therefore move freely through the scene. It is implemented by calculating the camera axis for movement in the right direction and using the lookat method to get the view matrix. Arrow Up, Down, Left, Right are for moving the camera and the mouse is used to look around.

Complex 3D Models from Files

For creating our whole map and some of the different object in the game we used Blender. We exported all the files as Collard (.dae)-Files to be able to retrieve some transformations of the different objects and to store light sources within the map file. Some of the objects we used in our game are not made by ourselves and therefore we provide the links the the original source:

- <https://www.turbosquid.com/FullPreview/Index.cfm/ID/1161465>
- <https://www.turbosquid.com/FullPreview/Index.cfm/ID/668299>
- <https://www.turbosquid.com/FullPreview/Index.cfm/ID/689819>

- <https://www.turbosquid.com/FullPreview/Index.cfm/ID/1091527>

To actually load all of the necessary 3D models in our game, we used the library Assimp.

Moving objects

The player's vehicle can be controlled, when the camera is following the vehicle, via keyboard input and it's tires are rotating, depending on the speed of the vehicle. All movement is simulated in a NVIDIA PhysX scene and transformation matrices for rendering are retrieved after each simulation step.

Hierarchical Animation Using Physics Engine

Hierarchical animations using NVIDIA PhysX are used to simulate the individually moving tires of the vehicle the player controls. The tires of the vehicle are four individual meshes that are children of the vehicle object in our tree-based rendering approach and the handling of the transformation of both the vehicle and the tires is fully done by the physics engine.

Texture mapping

Every object in the game has UV-coordinates, is therefore textured and is loaded with assimp from a resource file (.obj, .dae, etc). If a texture is missing (or not set for artistic reasons), a default white 1x1 pixel texture is used, so the color of the object is defined only by the material coefficients (ambient, diffuse, specular and emissive). The skybox is a cubemap wrapped on a simple cube, rendered as last object of the scene and with it's depth value set to max so it always fails the depth test if there are elements in front. The skybox texture was taken from:

- <https://93i.de/p/free-skybox-texture-set/>

Lighting and materials

The object loader reads all the light data from the resource file of the map (map needs to be saved as Collada-file therefore) and supports directional, point and spotlights. There is one directional light (moonlight), a few spotlights (street lamps) and some point light (to make the finish line more visually appealing) in the scene. All objects use the phong illumination model. Materials are loaded from the resource files via assimp and used for the lighting calculations.

View-Frustum Culling

For improving the performance of our game we implemented the view-frustum culling with the Bounding-Spheres approach. We therefore stored for each mesh the center point of the point of the mesh, that has the furthest distance to the center point. Then we calculated the front- and back-plane, and the four side planes of the view frustum pyramid in worldspace. We then calculate for every mesh, if the distance between the center point of the object and all the six planes is greater than distance between the center point and point with the furthest distance and if the calculated distance to each plane is negative (normals of each plane are

points inwards). If that is the case, then the mesh will not be drawn, because it is definitely outside of the view frustum.

Effects

GPU-Particle System

The particle system is implemented as an explosion effect with a following burn and cooldown phase. Every call to the emit function starts the effect on the given location, overwriting the old, if the effect is already started. A Particle consists of a vec4 for its position where the first three dimensions xyz are used as the position and w is used as it's time to live (TTL) and a vec4 for it's velocity, using xyz as the velocity vector and w as a texture id for rendering. Position and velocity of the particles are stored in a shader storage buffer, two for each to ping-pong between them every frame. When calling the emit function one emitter particle is written into the current active position and velocity buffers. This special emitter particle is ignored during drawing and it's only purpose is to spawn the actual particles for the effect during the update routine by running a compute shader. Every time the particle systems update method is called, the particles are read from the active buffer, written to the second one and then they are swapped. This happens fully on the GPU in a compute shader, so no bottleneck for synchronizing HOST and GPU is introduced. The compute shader distinguishes between emitter and usual particles by looking at it's TTL. Particles with a TTL greater than 10 are seen as emitters and spawn new particles, everything below is a usual particle which position and velocity needs to be simulated. The simulation process simply decreases the particles TTL by the time step and updates its position using Velocity Verlet integration[2] if it is still alive ($TTL > 0$). To fake a little air breeze, small random forces are added to the position. The emitter routine goes through different phases to create the explosion, fire and smoke effect, which are specified as uniforms to the compute shader. Depending on each phase, new particles are either spawned at random points in a hemisphere or circle. During the last phase the emitting particle is not written to the output buffer anymore to stop the system. Random noise values are calculated in the shader using a self contained implementation of Perlin Simplex Noise[3]. To keep track of the particle count for drawing, a atomic counter with a temporary buffer for reading is used. The particles are rendered instanced on a quadratic mesh which always faces the camera (billboarding). This is achieved by extracting the right and up vector of the view matrix and using it for the world position calculation of the particles in the vertex shader. The fragment shader draws the particles by using either a procedural generated black and white smoke or fire texture and tinting with a color.

Procedural textures

For the particle systems fire and smokes visual appearance cloud like textures are generated at game startup. The first step to this is a simple generation of a two dimensional, since we want to create 2D textures, noise array by using the rand function of the standard c-lib and normalizing it's return value to a zero to one range. Since this does not look like a cloud at all multiple "zoomed in" versions of the same noise texture are added together[4]. Each of the "zoomed in" versions is smoothed using bilinear interpolation to take neighbour

values into account. The more “zoomed in” versions are added together, the smoother the final texture gets. Our choices for this iteration count are simply based on how we liked the result. Since a texture is of quadratic shape, only a cut out circle is used in the final result. This is done by calculating the dot product of the local coordinate vector with itself and using this in an intensity calculation since we are also smoothing the edge of the circle. All pixels outside the circle will therefore be black.

Highscore persistence

After a successful run through the game the needed time through the course is saved as an integer in milliseconds with the current datetime as text into a sqlite database. The highscore table at the end of the game displays the top ten times needed to successfully complete the game. If the current run is under the top ten, it is marked with arrows on the left and right side of the line.

Subdivision Surface

The subdivision surface effect is used in general to smooth out a rough base mesh, by splitting up edges, faces and/or vertices and creating new faces. But splitting up faces/edges/vertices does not quite do the job, so both the old and the newly created vertices have to be repositioned to make the resulting surface smoother than the base mesh. To get even better results, this process can be applied to a base mesh iteratively, resulting in the base mesh converging to a limit surface, when applying the algorithm an infinite number of times.

There are a lot of different approaches and different algorithms for accomplishing some kind of subdivision surface effect, like the Catmull-Clark Algorithm, the Doo-Sabin Algorithm, the Butterfly-Scheme, the Loop-Scheme and many more.

To decide, which of those just mentioned to implement, we had to take a look on number of vertices each of our polygons have. Since loading our objects with assimp resulted in them being fully triangulated, we could assume that all of our base meshes would consist of triangles only. Furthermore OpenGL works with triangles as well. We therefore decided to implement the Loop-Scheme, since it works (only) triangle meshes, it always gives a smoother triangle mesh as result and is fairly easy to implement.

The Loop-Scheme basically smooths out a base mesh by splitting each triangle of the base mesh in four new and smaller triangles, by inserting new vertices between each pair of vertices in a triangle. The newly created as well as the old vertices are being repositioned by a fixed scheme. After the new mesh has been created, the Loop-Scheme can be applied to it again, for getting an even smoother result.

The first step in our implementation of the subdivision surface effect is to create some kind of adjacency information for each vertex of our mesh. Since the assimp-object loader results in having split up each vertex in the number of adjacent faces, and therefore losing pretty much all the available adjacency information of the vertex, we had to recreate it ourselves when loading the mesh. We stored the resulting vertices in a form, that fits our needs for later processing on the graphics card, with all the adjacency information needed. We therefore stored the vertex information (position, normal, uv-coordinates) together with the number of adjacent vertices and an offset, that refers to an offset in an adjacency list we created as well.

Furthermore we created an edge table, that contains the information for each edge, which vertices are directly connected to this edge (always two) and with vertices are connected to this edge via faces (up to two, because each edge can only have a maximum of two adjacent faces). The initial idea how to store the the necessary information in a suitable way for the graphics, we gathered from the article [1].

After we have done all the preprocessing and had our information stored and ready to use, the actual subdivision of the mesh has to be done. In each frame we check, if the subdivision level (one subdivision level = one iteration of the subdivision surface effect) has changed. If it has, the mesh is being recalculated from the base mesh. With the available information, we can transfer it to the graphics card via SSBOs and do the recalculation and reconnection of all the vertices in parallel for all the vertices at the same time using a compute shader. After we have calculated the new mesh, we have to create a new edge table for the subdivided mesh, in case the we have to do another iteration, which is done by another compute shader.

After the hole processing of the mesh is done, the result of the subdivision (a buffer of vertices and another buffer of indices) gets bind to the vertex shader, and can the used to render the object, so the subdivision does not have to be done every frame.

In our game the subdivision level can be controlled ingame with the 'F5'-Key (minus one subdivision level) and the 'F6'-Key (plus one subdivision level). The minimum subdivision level is 0 (no subdivision at all) and the maximum is 3 (three iterations). The subdivision surface effect is not applied to every object in our game, but can be seen on the street lights and trees on the map.

Important note: When changing the subdivision level, all objects have to be processed, which leads to a short lag, especially when processing the objects with subdivision level 3.

Another important note: When view-frustum culling is turned on and the subdivision level changes, only the objects inside the view frustum are being processed, which results in a shorter initial lag, but also means, that when the camera moves and new object are getting inside the view frustum, these objects have to processed as they enter the view frustum and cause another short lag.

Lightmapping

For the mandatory light mapping effect we decided to display shadows (mainly casted by the directional moonlight), since our game does not support shadow mapping and it is therefore a very good extension of our current rendering engine, to show more detail. In our game the light mapping effect has only been added to the street textures because it is best visible there and gives the best effect.

Controls

Key	Effect
Arrow Up	Accelerate

Arrow Down	Reverse/Brake
Arrow Left	Steer left
Arrow Right	Steer right
Space	Handbrake
ESC	Quit game
F2	Frametime on/off
F3	wireframe on/off
F4	Vehicle explosion on collision on/off
F5	Subdivision Surface lower level (min. 0)
F6	Subdivision Surface higher level (max. 3)
F7	Activate fire particle system
F8	View frustum culling on/off
+/~	Toggle fullscreen/windowed
C	Toggle camera freeview/follow
R	Restart the game

Arrow Up, Down, Left, Right, Space, + and C key are polled every frame and F1-F8 and ESC are implemented using key callbacks.

Gameplay

The players vehicle can be controlled via the keyboard and steered through the street of the map while the camera always follows the vehicle by keeping a small distance. The vehicle behaviour is simulated via the vehicle implementation of PhysX. Since the player plays a student who loves to race, but has no money to buy himself an own car, he has “borrowed” his father’s car (without him knowing of course). It is there most importantly - besides getting a good time - to not crash the car at all. When the player hits anything in the map, the game is instantly over, because the father of the player must not find out, that he “borrowed” his car.

Note: Since the it is not allowed to hit anything with the car, it is important to brake and drive a bit slower because as soon as the car chassis hits anything, the game is over. For testing purposes there an option to be invincible to car crashes, that can be toggled with the ‘F4’-key. But when the car flips over the game is over anyways because the vehicle cannot flip itself over again and the game has to be restarted with the ‘R’-key.

Adjustable Parameters

Screen Resolution

The resolution of the game changes when the window is resized by calculation a new resolution which keeps the games fixed aspect ratio of 16:9. Window resolutions out of this ratio will have black bars on the top/bottom or left/right. If the window size changes, the viewport and the projection matrix are changed and recalculated. The initial starting screen resolution can be set in the settings file.

Fullscreen/Windowed Mode

Fullscreen/Windowed mode can be toggled with the +/- key on the keyboard or initially set in the settings file.

Refresh-Rate

Defaults to 60Hz and can be changed via the settings file. The implementation uses `glfwWindowHint(GLFW_REFRESH_RATE, refresh-rate)` for setting the refresh rate at game start.

Brightness

Can be changed via the settings file and is implemented by incrementing the ambient lighting of every object in the scene.

Used libraries

- NVIDIA PhysX for collision and vehicle simulation
<https://developer.nvidia.com/physx-sdk>
- GLM for math
<https://glm.g-truc.net>
- ASSIMP for model loading
<http://www.assimp.org/>
- STB Image for image loading
<https://github.com/nothings/stb>
- Inih for settings file parsing
<https://github.com/benhoyt/inih>
- Freetype for font loading
<https://www.freetype.org/>
- GLFW for window handling
<http://www.glfw.org/>
- GLEW for opengl extension loading
<http://glew.sourceforge.net/>
- SQLite for highscore persistence
<https://www.sqlite.org/index.html>

Literature

[1] Feature-Adaptive Rendering of Loop Subdivision Surfaces on Modern GPUs

Huang, Yun-Cen ; Feng, Jie-Qing ; Nießner, Matthias ; Cui, Yuan-Min ; Yang, Baoguang
Journal of Computer Science and Technology, 2014, Vol.29(6), pp.1014-1025

[2] Verlet integration:

<https://www.saylor.org/site/wp-content/uploads/2011/06/MA221-6.1.pdf>

[3] Simplex Noise: <http://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>

[4] Random Noise: <https://lodev.org/cgtutor/randomnoise.html>