

Space Racer

Florian Wicher (01526729), Juliette Dubois (11831465)

Features

- Realistic behavior of the space vessel and asteroids, thanks to physics engine
- Glittering asteroids (vertex normal mapping)
- An awesome, procedurally generated planet in the background

Controls

The space ship of our hero travels at a constant speed (although he can go faster for a limited time after collecting an extra that allows him to).

- W,A,S,D: steer the spacecraft (Easy way to control the direction. Allows tutors to easily verify that the win-condition is working.)
- I, J, K, L: fire thrusters (Hard way to control the craft. The way real astronauts play this game.)
- Space: Impulse forward
- B: Impulse backwards
- O,P: Increase/decrease brightness
- Q: Fire regular shots
- E: Fire self-guided shots

Gameplay

You need to shoot all the asteroids in the time you have. The interface indicates both the numbers of asteroids left to shoot and the amount of time you have left. Should you run low on time, you can collect the extra that is situated between the two pillars in the scene. After you have shot all asteroids, fly in between the two pillars in time to finish the level.

Light

We are using the lights from the ECG-Framework – a directional and a point light.

Texture

Textures are applied for the lens flare and the objects in the game using the cg frameworks texture class.

The asteroids are a solid gray, but use normal mapping, whence the craters and the surface structure.

The particle system generates the flames in its geometry shader and calculates the color of the flames as a function of the distance from the fire's center.

To make the hud, a simple square is drawn on the screen and filled with different textures.

Used libraries

<http://www.assimp.org/> for loading obj files

<https://www.nvidia.fr/> nVidia Physx for calculating collisions.

Camera

The camera remains fixed behind the vessel.

Which tutorials did we use, and for what?

C++ Primer by Stanley Lippmann: Countless of the techniques described, e.g. random number engines and distributions, smart pointers, inheritance in C++, lambda functions, insert-iterators, etc. etc.

OpenGL Superbible: Used as a refresher on OpenGL.

Grundwissen Mathematikstudium (Arens et al.): Gram-Schmidtsches Orthonormierungsverfahren (S. 670), Kapitel 7: Analytische Geometrie (S. 234ff.);

Textures in OpenGL : <https://www.youtube.com/watch?v=n4k7ANAFslQ>

Tutorials on OpenGL with C, especially the tutorial on billboarding and vertex buffer feedback for the fire: <http://ogldev.atspace.co.uk/>

A number of approaches for frustum culling: <http://www.lighthouse3d.com/tutorials/view-frustum-culling/view-frustums-shape/>

Lens flare (Java): <https://www.youtube.com/watch?v=OiMRdkhvwgq>

Normal Mapping: <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

Transformation of surface normals from tangent to world space (normal mapping): <http://www.terathon.com/code/tangent.html>

Vertex shader animation: Slides from the CG lectures at TU Wien

Procedural textures: <https://thebookofshaders.com/11/?lan=fr>

PhysX:

<https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/Index.html>

<https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXAPI/files/index.html>

Techniques we implemented

1. Basic gameplay:
Win-loose-condition. If you manage to fly in between the goal posts within 40 seconds, you win. Else, you loose.
2. Physics
Implemented collision detection between asteroids, and asteroid and ship. Based on the indicated bounding shapes and initial movement and torque, the framework calculates the interaction between the objects. The movement of the ship is also realized by sending commands to the PhysX Framework.
3. View Frustum Culling
Each object has a bounding sphere, if this sphere is outside the view-frustum that is composed of 6 planes (in the negative half-space of at least one of the planes), the object isn't rendered. See console output for number of draw calls intercepted by frustum culling.
4. HUD
To make the hud, a simple square is drawn on the screen and filled with different textures, each one corresponding to one part of the hud. The hud is used to show the time, and also the win / loose screens.
For the chrono each seconds a function is called to change the textures drawn in the square.
5. GPU Particle System using Transform Feedback

The fire on the left goal post (we didn't know where else to put it) is realized using transform feedback. The positions of the flames of the fire are generated by using a `random()` function inside the `UpdateParticleSystem` geometry shader that uses the position of previous flames in the environment as a seed; then rendered as triangles.

6. Vertex Shader Animation

In the vertex shader, there is a uniform for the time. This uniform is used to move the vertex position.

Then the normals need to be adjusted to the new position. Since the surface before transformation is a cylinder we can compute explicitly the new normal by taking the cross product of the derivative of the surface after transformation.

7. Procedural textures:

The resource given above explains how to generate random number and 2D fractional noise (adding several octaves with various frequencies and amplitudes). In the resource a function is proposed that take a 2D vector input and return a float as noise value.

To create the texture of the planet, this function is used two times for each fragment.

First, it is called with the (x,y) coordinates of the vertex as an argument (z was not used). The resulting noise is used to take a value from a color map ranging from red to orange. The number of octave, frequencies and amplitudes were adjusted to obtain a planet-like texture. Then the function is called with (x,y) position multiplied by time as an argument. This is used to make a kind of moving texture, like some clouds on the surface of the planet. Fewer octaves were used to have a less precise texture. This noise is used to take a value between black and white.

The color of the fragment is then computed as the sum of the color of the planet and the color of the cloud.

8. Simple Normal Mapping:

The asteroids surface structure is achieved using normal mapping. Normals are responsible for how light is reflected off a surface. By altering just vertex normals, one can model intricate surface structure without having to generate complex geometry. This approach saves space and consumes less processing power.

9. Lens Flare

A light source position is indicated (the fire in our case), then the flare-textures handed to the Flare manager are distributed on the ray that originates in the screen center and passes through the light source. The opacity of the flares increases as the light source comes closer to the center of the camera.

What tools did we use?

- Visual Studio
- Nvidia nSight
- JetBrains ReSharper
- Blender (for creating the models)
- Photoshop and Plug-ins by nVidia to create textures and generate normal maps.