

3D Asteroid Game

A 3D game developed in C++. The main goal of this game is to reach Earth with the spaceship and avoid collisions with asteroids.

Player has 10 lives, so if the plane crashes to asteroids, player will lose 1 life. And the player will be for 2seconds invulnerable.

The spaceship can collect stars to refill the fuel.

First feedback:

Mouse is disabled in the game based on the first feedback.

Bloom effect is corrected.

Playability: The player has to reach the earth to win. Player has 10 lives, if plane crashes to asteroids, lives will be reduced.

Also Fuel is limited and player can collect stars to refuel.

Run.bat: Game is adjustable through this file. (fullscreen, brightness, etc.)

Legal Notice

The game uses the following libraries:

1. GLEW - MIT (<https://github.com/nigels-com/glew#copyright-and-licensing>)
2. GLFW - zlib/libpng, a BSD-like license (<http://www.glfw.org/license.html>)
3. GLM - MIT (<http://glm.g-truc.net/copying.txt>)
4. tinyobjloader - MIT license (<https://github.com/syoyo/tinyobjloader/blob/master/LICENSE>)
5. stb_image - public domain (<https://github.com/nothings/stb>)
6. freetype - FreeType/GNU GPL2 (<https://www.freetype.org/license.html>)
7. bullet - zlib (<https://github.com/bulletphysics/bullet3/blob/master/LICENSE.txt>)

The models are from the following sources:

1. star wars spaceship model - <https://clara.io/view/c7cf473c-45af-4245-aa97-308aa1a5dd75>
2. pbr materials - <https://freepbr.com>

Code samples are from:

1. <https://learnopengl.com>

Game Controls

The spaceship can be controlled with the mouse and the WASD buttons.

- **W** Forwards
- **S** Backwards
- **D** Right
- **A** Left
- **Q** Upwards
- **E** Downwards

- **I** move Sun forwards
- **K** move Sun backwards
- **L** move Sun right
- **J** move Sun left
- **O** move Sun upwards
- **U** move Sun downwards
- **F2** show/hide FPS
- **F3** toggle wireframe mode on/off
- **F4** toggle collision on/off
- **F5** toggle fuel consumption on/off
- **F7** toggle bloom on/off
- **F8** toggle frustum culling on/off

Requirements

Everything from above in the Legal Notice

Tested with Intel HD Graphics 3300, Intel HD Graphics 4300 and NVIDIA GeForce 820M

Implementation

Effects

Light mapping

For light mapping, I chose to use the nowadays really popular physically based rendering (PBR) method. In PBR, one of the properties to use during the fragment shader stage is an ambient occlusion (AO) texture. This AO texture shades parts of the object based on the vertices' texture coordinates and thus maps lighting on the surface of the object.

All object are illuminated by the „Sun“ which is an invisible point light source at the point (100, 100, 10).

Physics Engine

For the physics engine I used the open source ~bullet~ physics engine for collision detection and animation of the asteroids. The ship and asteroids are the only objects that could collide with each other, the stars and Earth couldn't. The asteroids have bounding spheres while the spaceship has bounding box for collision detection.

To animate the asteroids, I apply a torque impulse on the object, which rotates it around its axis, and then with every iteration of the game I get the current rotation quaternion from the engine.

Heads-up Display

The Heads-up Display (HUD) is handled by rendering text with the freetype open source library.

GPU Particle System

The GPU particles have position and random velocity attributes. OpenGL transform feedback is used to get back the updated position attributes of the particles.

Bloom

RGB values above a given threshold are captured in a separate framebuffer. Gaussian blur is applied on that framebuffer and the HDR and blurred images are combined to make the Bloom effect visually appealing.

Loading Complex Objects From File

The game uses the tiny-obj-loader open source library, which has a single file header-only implementation. It can be easily integrated with projects this way and could load .obj and .mtl files. To import textures I used stb-image, which is again a single file header-only library. For every model I defined its buffers, a vertex array object and vertex buffer objects for position, normal and texture coordinates. These buffer objects are used for rendering the models.

Animation Using Physics Engine

Only the asteroids are animated. Using the bullet physics engine a torque impulse is applied on creation of the asteroid, and for each iteration the current rotation is queried from the physics engine, and is applied to the MVP matrix when rendering the object.

View-Frustum Culling

Simple 3D geometry is used to calculate which object falls within the view pyramid. This is done by calculating four direction vectors (four corners of the screen / camera). By knowing these directions, one could determine the planes that define the view frustum, and thus calculate whether an object is inside the pyramid the camera can see. The frustum culling function also calculates the distance from the planes, and if the distance is less than the radius of the object's bounding sphere, or the object falls inside the pyramid, the program renders the object.

Also, a simple 3D vector calculation gives back which object falls behind the camera, eliminating unnecessary calculations for the frustum culling. This is done by a dot product of the camera front vector and the camera-to-object vector. If the dot product is negative, meaning the angle between the vectors is greater than 90 degrees, the object is behind the camera.

Debug Options

F3 enables wireframe, which is a simple OpenGL call.

F4 collision can be on/off.

With **F7** bloom can be enabled. This is done by specifying the framebuffer that is being used and by limiting the number of blurring iterations on the bloom image.

Frustum culling can be enabled/disabled by pressing **F8**.