# Kürbislutscher

Katharina Unger (1325652) und Markus Lehr (1426019)

## Controls

| Key | Effect |
|---|---|
| W,A,S,D | Horizontal and vertical movement. |
| [Ctrl] | Shoot laser |
| [Shift] | Speed boost |
| [Space] | Tongue animation (licking) |
| Arrow Keys | Navigate through menu |
| [Enter] | Select menu entry |
| [Esc] | Exit game |
| | |
| [F2] | Frame Time on/off |
| [F3] | Wire Frame / Scene with collision shapes |
| [F4] | Texture-Sampling-Quality: Nearest Neighbor/Bilinear |
| [F5] | Mip Mapping-Quality: Off/Nearest Neighbor/Linear |
| [F6] | Bloom on/off |
| [F7] | Motion Blur on/off |
| [F8] | Viewfrustum-Culling on/off |
| [F9] | Blending on/off |

Calling the exe by double clicking starts in fullscreen mode. Calling it with the parameters [width] [height] [fullscreen:0/1] will set the resolution and whether the game should be in full screen mode or not.

# Implementation

As a basis for our project we had a basic implementation from a previous try, which only covered the tutorial steps. Our objects and camera are structured in a scene graph, which also updates the relevant bullet collision shapes. The modelloader can either load animated or static meshes, which unfortunately have separate rendering processes, since our static rendering pipeline and shaders did not work with bone information.
The game is controlled in a game manager, which handles events, such as loading levels, losing lives or displaying score. Our sound effect class is also placed here, so that all other objects have a reference to our audio sources.
Initial positions of asteroids and pumpkins are generated in a level generator, which returns a 3D array, representing the level. According to that array and to hard coded parameters, lights and other objects are positioned in the scene graph initialization function.
The drawing process recursively draws each object and then it's children, starting from the scenegraph. The generated image is then altered with a bloom and motion blur shader and finally rendered. Outside of our graph structure, we draw the 2D elements afterwards, which essentially are 3D cubes with just one side textured, rotated and positioned at the right position.

# Features

The main difference between our game and others is that levels are procedurally generated with just a handful of parameters. Levels are never the same and ramp up in difficulty as the player proceeds on.
Also, all our shaders are parameterizable, which allowed us to eg. increase the motion blur effect only when the player is boosting.
Additionally to the necessary features, we implemented small touches here and there, which make the game seem more natural. The spaceship, for example, shakes a little when it hits an asteroid.

# Lighting

Simple ambient and specular lighting for all objects.
Dynamic lighting with moving light sources, namely each individual laser beam serves as a light source that illuminates the pumpkins and asteroids it flies by.
Dynamic lighting with light source at the end of the Spaceship.

# Textures

For importing objects and textures assimp was used. The parallel data structure that we are using saves the meshes, UVs and normals in vectors and saves them to the designated gl buffer.

The hud and menu (2D objects in general) are using the same data structure and modelloader as 3D objects and are represented by a 1x1x1 cube with the designated texture

on only one side of it. The reason for that is, that the performance loss and memory gain is negligible and implementing a 2D rendering pipeline was not time efficient.

We are not using a text library, but store text and icons in individual, square png images and use them in the material file of the gui element's .obj file. (The reason is - again - time efficiency)

## Effects

Bloom (on laser-beam): implemented following the tutorial from https://learnopengl.com/#!Advanced-Lighting/Bloom

Motion Blur: combination of http://john-chapman-graphics.blogspot.co.at/2013/01/what-is-motion-blur-motion-pictures-are.html and https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch27.html

GPU Vertex Skinning: No tutorial has been followed. The assimp modelloader was used to copy the data structures and store bone information. The assimp animator was used to get the interpolated bone positions, which are then stored in the shader's array buffer.

## Tools

All models despite the tongue were created using Blender. The Tongue and its animation was created with Maya.

## Libraries

Libraries in use are:

- Bullet (http://bulletphysics.org/wordpress/)
- Assimp (http://assimp.sourceforge.net/index.html)
- Freeimage (http://freeimage.sourceforge.net/)
- irrKlang (http://www.ambiera.com/irrklang/index.html)