

# ToyWars

Takaki Hagen(11701261)  
Christoph Wottawa(0527408)

“ToyWars” is a single player shooting game. You have to shoot down enemies as fast as you can.

## Features

Single Player Shooting game  
Real Video on the TV Set in the room!  
Fully GPU-animated particle system  
Animated Water in the fishtank

## Gameplay

The player controls a drone which is in the center of the screen. There are enemies in the level which hover(they don't move) and try to shoot down the drone. The player has to dodge the bullets from the cannons and try to shoot them down as many as they can. If the player hits the enemy, he will get 10 points, and a new enemy will emerge. The goal of this game is, to get as many points as you can.

## Controls

With the arrow keys, you can move the drone to up/down and to right/left and with WASD forward/backward and rotate to right/left. You can also shoot with F key or the space bar. If you don't hold any key, then the drone will only hover and won't move. You can quit the game with esc key.



Key	Move
mouse	move the camera direction
WS	move to forwards/backwards
AD	rotate to right/left
Space Bar	shoot
R	restart the game(only by game over)

## Technical Details

Our game is a simple shooting game. The player(drone) can move forward, backward, right or left, to where the camera looks at. The camera can move with mouse and objects, which are not in the camera view, will be culled. We implement frustum culling with bounding sphere of each object. The Physx are used to calculate the positions of bullets and player, and to detect collision of objects. We generate a room, player, enemy and bullets. For player, enemy and room, we used complex 3D models from files. The drone has 4 propellers in child object, which rotate with hierarchical Animation. Also, as HUD, we used FreeType library and display information of Game. Effects are explained below. All objects in the scene are textured and illuminated. There are 4 distinct types of objects, that have to be rendered:

### Type of object in the game

The drone – Animated mesh

The room – a simple static mesh

The cannon balls – Dynamically generated meshes

Particles – GPU Particle System (Transform Feedback, Instancing)

### HUD

Displaying number of drawing objects, frame time in millisecond, FPS, ranking, and current point. In case it is, game over, the text “Game Over” is also, displayed.

#### link of reference page

- <https://learnopengl.com/In-Practice/Text-Rendering>
- <https://www.freetype.org/>

### Model

All Models are created or modified using Blender. The room is created with an architecture toolkit called archipack, which makes it easy to create walls, windows, floors and ceilings. Models for the Quadrocopter are from the ‘net and modified to animate the rotors. Enemies got a model of a Star Wars Death Star.

#### link of reference page

- <https://www.youtube.com/user/AndrewPPrice>
- <https://free3d.com/3d-model/puo-4037-34535.html>
- <https://www.turbosquid.com/FullPreview/Index.cfm/ID/1128793>

### Light mapping

Light maps are generated in Blender. The whole scene has two light sources: a lamp at the ceiling and the sun outside. Every texture is baked with the Cycles Render Engine, then the

resulting texture files are re-applied to the surfaces of the model. The room model is then exported to OBJ format, with references to the baked texture files.

### Physics Engine

Using PhysX Library from Nvidia to calculate the position and collision of objects. The moving of bullets are calculated and thus are animated. Transformation matrices from the PhysX Actors are applied to the geometries of the bullets to simulate the “throwing of a ball”.

Calculated object

- Bullet from player
- Bullet from enemy
- Drone(player)
- Wall

link of reference page

- <https://developer.nvidia.com/physx-sdk>

### Camera

The Camera is always looking at the drone. Camera is moved by mouse.

### Illumination

There is one central Light Source - a Lamp hanging on the ceiling, that illuminates the room. This is a simple point light source

### Collision detection

Collision detection is done with the help of nVidia PhysX.

### Debug Options

For some function, we have implemented debug options. If the debug option is activated or deactivated, the message will displayed on command line.

Key	Effect
F2	Activate/Deactivate Frame Time Display
F3	Activate/Deactivate Wireframe
F4	Activate/Deactivate backface culling
F8	Activate/Deactivate frustum culling

### Effects

- Procedural textures
- Video textures
- Shadow mapping
- GPU-Particle System (+Transform Feedback,Instancing)

Name	Effect1	Effect2
Takaki Hagen (2.5 effect points)	Shadow mapping with PCF (1.5 point)	Procedural texture (1 point)
Christoph Wottawa (2 effect points)	Video texture (1 point)	GPU-Particle System (+Transform Feedback,Instancing) (1 point)

## Implementation of Effects

### 1. Procedural textures

In our Game, the picture of a marble on the wall is generated with this method.

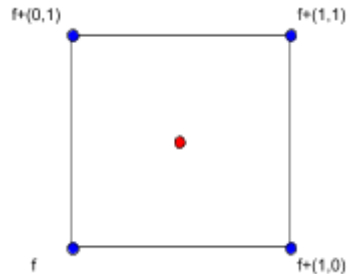
Procedural textures are generated in fragment shader in following processes.

1. Make several different scale of "Perlin Noise"
2. Add all these noise and take the average of these each values
3. combine the noise from 2. with the function sin like below  

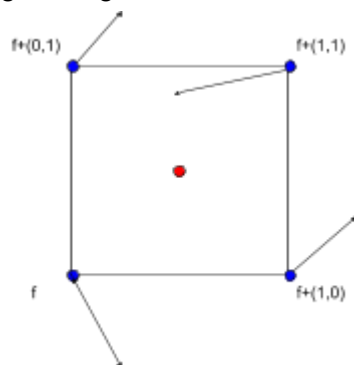
$$(1 + \sin((uv.x + texColor.x/2) * 50))/2$$
4. Draw the calculated color on target object

Perlin noise

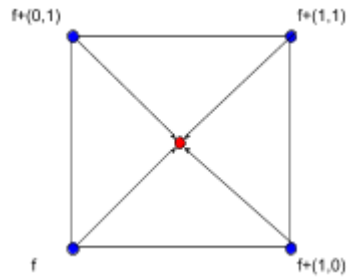
1. calculating  $f = \text{floor}()$  of input position
2. make a box like below. The red position is the input point



3. get the gradient vector with pseudorandom function



4. get the distance vectors like below.



5. calculate dot product of gradient and distance vector to get the noise value

#### link of reference page

1. <https://thebookofshaders.com/10/>
2. <http://math.hws.edu/graphicsbook/c7/s3.html#webgl3d..3>
3. <http://lodev.org/cgtutor/randomnoise.html>
4. [www.upvector.com/?section=Tutorials](http://www.upvector.com/?section=Tutorials)

## 2. Video textures

Video textures are read via FFMPEG. The video file is read and split into its streams (audio and video) - only the video stream is used. The video stream is processed frame by frame. Every frame needs to be converted from YUV color space to RGB, which is done using FFMPEGs `sws_scale()` function. To use this texture on any surface and any `Material()` Object, a `Texture()` object gets instanced with an image and its OpenGL Texture Identifier is used to bind a new buffer and write the video frames to. For each frame mip maps are generated via `glGenerateMipmap(GL_TEXTURE_2D)`.

## 3. Shadow mapping with PCF

Shadow mapping are done by following processes.

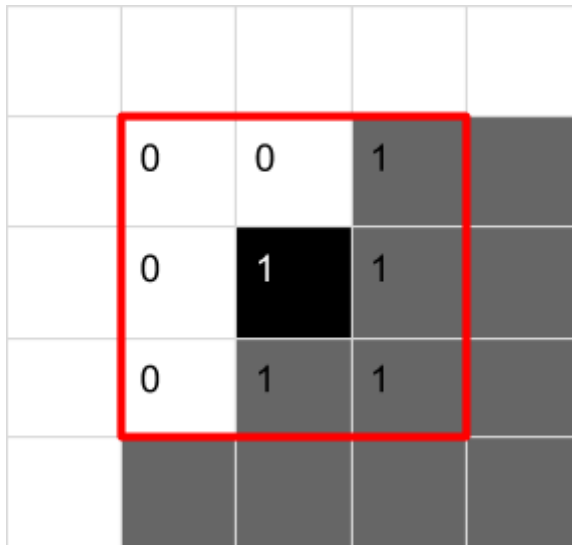
1. Draw Depth map on depthmap texture. In this part, we applied orthogonal camera to draw depthmap.
2. Distinct whether the handling point is behind the nearest object to light sources or not.

The shadow of dynamic Object, such as drone, enemy and bullets are drawn with shadow mapping in our game. In this game, shadow of directional light are drawn. In our project, the difficult part was to remove the shadow acne completely, so that the peeter panning doesn't occur. For the enemy and bullets, we did the front culling to draw the depth map. In addition to front culling, shadow bias is applied for drone. Because of thin part, like propeller, we have to apply the shadow bias to avoid shadow acne.

### PCF

In the fragment shader, the program checks, whether the handling points are shadow or not. To Generate Softer shadow, the shader, checks whether the nearest posits are shadow too or not and calculate average values. In the picture below, if we

handle the one black pixel, the value will be  $(0*4+1*5)/9=0.556$ . Thus, the shadow will be softer.



link of reference page

1. <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>
2. [www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/](http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/)

#### 4. GPU-Particle System (+Transform Feedback,Instancing)

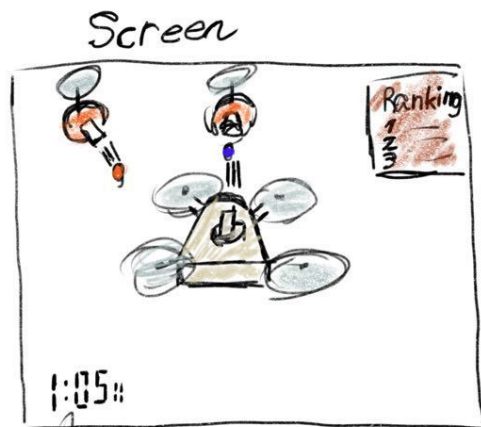
For Transform Feedback we need two Buffers, which are bound to the TFB alternately. One contains the current particle data (position, velocity, lifespan) and the other is used to store the results of the transformation calculation in the TF shader (we need two different programs, one for TF, one for drawing). The results are then used for drawing with the second shader. For the TF Shader drawing is disabled, since we only want the results of the shader for our program and not on the screen. (`glEnable(GL_RASTERIZER_DISCARD)`).

Instancing gives us a nice tool to save bandwidth between the CPU and the GPU, since the GPU is going to instantiate hundreds or thousands of the same mesh for us. One mesh (for our particles) is sent to the GPU - a VAO and VBO is created, the VBO with just the vertices for one particle. The results of TFB contains position data for every single particle that is to be instanced. Calling `glDrawArraysInstanced()` is used for drawing.

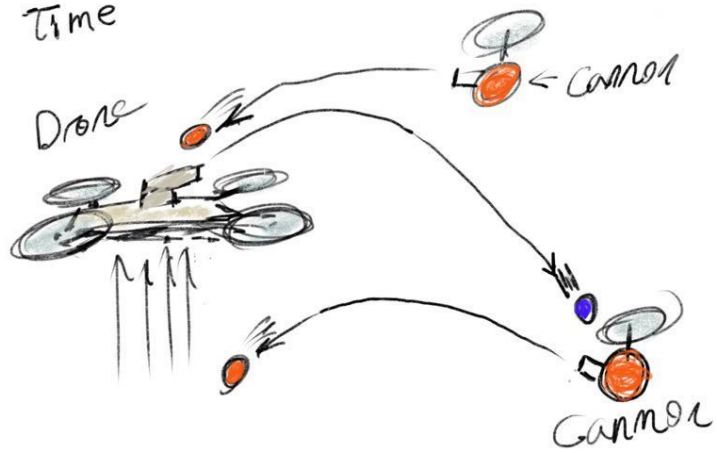
reference pages:

- <http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html>
- <https://learnopengl.com/Advanced-OpenGL/Instancing>
- <https://open.gl/feedback>

## Sketches



Time



Shooed down

