



## Documentation

### Kart Racer Madness

CGUE SS 2018

Kurt Christian Chabek 09901743

Thomas Steinschauer 01426150

### Game controls and config

Set window resolution, fullscreen etc. in **bin/config.ini**.

, and . (**comma and period**): control **brightness**

Press **F1** in-game to see all available controls!

Press **F10** in-game to activate the **Debug-Controller** (fly around with WASD / Mouse)!

Change the **number of laps** in **assets/game.ini**.



## Overview

Kart Racer madness is a split-screen racing game.

The scene is lit by one directional light, and we implemented these effects:

- Water mesh
- Normal mapping
- Separate Lightmaps
- Shadow Mapping with PCF
- CPU Particle System with Instancing

## Implementation

### Overview

There are multiple projects in the solution:

CGUELib: The Engine

Game: The actual game

Playground-Chris, Playground-Thomas: Playground and test-area for the two developers

The engine is heavily inspired by Unity (<https://unity3d.com/de/>) and Urho3D (<https://urho3d.github.io/>).

An object in the game is called Entity, and its components control appearance and behavior. The Transform-component for example provides position and orientation, a MeshRenderer-component renders a mesh.

## Hierarchical animation

Parent-child relations are set via the Transform-Component.

## Model Loading

The whole scene is modelled in Blender, and we import meshes, lights, materials, and textures. (Plus we post-process special objects like triggers, colliders, etc.).

See:

Scene loading: *CGUCLib/Scene::load*

Post-processing: *Game/functions.cpp*

## Car physics

Car physics are based on this tutorial: <https://www.youtube.com/watch?v=LG1CtIFRmpU>. The most important thing about the physical behavior of the cart is the suspension. For every vehicle there are four points that define where the wheels are located. In every frame four rays are cast from the wheelpoints downwards. If the ray hits the ground a force is added to the cart in the upwards direction. The larger the distance, the greater the force. This creates a kind of “bouncy-effect”. To avoid endless bouncing a friction-force is added to the car as well.

Acceleration and steering is only activated if the cart is grounded. That means that at least 3 of the four raycasts hit the ground. The acceleration simply happens by adding an acceleration force in the forward/backward direction. Steering is implemented by adding a torque force to the vehicle. To ensure the car only moves in the forward direction a grip force has to be added. The general drag force simulates general friction coming from ground or air.

See: *Game/Components/Car*

## Frustum culling

For frustum culling we test AABBs against the camera frustum in world space. A frustum can be constructed from either camera parameters (near, far, fov, aspect ratio) or from a view-projection matrix (by applying the inverse to a cube ranging from -1 to 1). We got more stable results with the first version.

AABBs are constructed from meshes and then transformed into world space, by applying the world matrix to all corner-points and rebuilding the min/max values.

See:

*CGUCLib/Graphics/View::updateDrawables*

*CGUCLib/Math/Frustum*

*CGUCLib/Math/BoundingBox*

References:

<http://www.lighthouse3d.com/tutorials/view-frustum-culling/>

## Camera movement

Soft camera movement is achieved with quaternion interpolation. Some “hacks” were necessary to keep the camera stable on x- and z-axes.

See: *Game/Components/FollowCam*

## Particle system, instancing, animated textures

Particles are spawned randomly in a cone. The angle and the rotation of the cone is adjustable. You can also change the emission-rate, the start-speed, the start-lifetime, start- and end-alpha-value, start- and end-scale such as the texture of the particles. It is also possible to randomize the emission-rate and the scale. Particles are simple quads that always look to the camera. Some Particles are animated using a grid-like texture. Depending on the rest-lifetime of a particle, the right part of the texture is chosen for rendering. You can also choose if the particles system loops over and over or if it stops emitting particles after a certain time. The particles are rendered using instancing. The whole simulation takes place on the CPU and the necessary per-particle-information (position, alpha and scale) is sent to the GPU via attributes.

In the game the rolled up dust of the drifting car is visualized with particles. Depending on the “drift-level” the emission-rate is raised or lowered. If the cart falls into the water, you can see a watersplash. This is also done using particles. Every time the car hits the watersurface, a particle system is spawned. The car-boost is also visualized using particles. Here you can see animated textures, so that every particle does not look the same. Every time the boost is activated, flames and smoke come out of the car.

## Shadows

Shadows are implemented with shadow maps. We prevent **shadow acne** by applying a small depth bias (configurable per light), **PCF** is used to reduce blocky edges.

The engine supports multiple shadow-casting lights. (The scene has only one light though.)

This page describes very well the steps necessary to get nice shadows:

<https://gamedev.stackexchange.com/questions/73851/how-do-i-fit-the-camera-frustum-inside-directional-light-space>

We calculate the shadow projection box based on the camera frustum, but with a reduced far plane to keep the shadow quality high. This works in our case, as the scene is lightmapped and only the cars and a few dynamic objects cast shadows. But you can see some of the problems mentioned in the link, especially the “swimming” shadows, as the projection box changes every frame when the camera is moving.

See:

*assets/core/shaders/lighting.inc*

*CGUCLib/Graphics/View::update*

References:

[https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx)

## Watermesh

Seven waves can be overlayed to simulate Water. For every wave you can set parameters such as, speed, amplitude, direction and phase. The meshdata is a simple grid and the waveinformation is passed to the shader using uniforms. The positions, tangents, co-tangents (for normal-mapping), and normals are computed analytically in the vertexshader.

## Normal maps

The whole scene is normal mapped (we import normals, tangents and bitangents from Blender). The water uses two scrolling normal maps to achieve this “small-waves”-effect.

## Lightmaps

Lightmaps were generated in Blender as separate textures per mesh. They are applied using a second uv-set.

See:

*assets/core/shaders/lighting.inc*  
*CGUELib/Core/Scene::load*

## More effects!

### Soft water edges

We do a depth pre-pass, then use this depth map to calculate the distance from water to opaque objects and apply transparency accordingly. The water also fades out over distance.

See: *assets/shaders/water.frag*

References:

<https://www.youtube.com/watch?v=qgDPSnZPGMA>

### Physically based shaders

The default shader uses the Blinn-Phong lighting model in combination with Fresnel and a normalization factor. There is one very nice reflection in the game when driving up the last hill – see image on next page.

See *assets/core/shaders/lighting.inc*

References:

Real-Time Rendering 3<sup>rd</sup> edition, pp. 233 and pp. 257



## Used libraries

### **Assimp**

FBX loading

<http://www.assimp.org/>

### **FMOD**

Audio

<https://www.fmod.com/>

### **FreeImage**

For loading images

<http://freeimage.sourceforge.net/>

### **FreeType**

Text

<https://www.freetype.org/>

### **PhysX**

All physics related stuff

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



