Developing a Jump and Fly Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Jan Dreier[*] Denis Golovin[†] Moritz Ibing[‡] Simon Grätzer[§] Jan Garcia[¶]

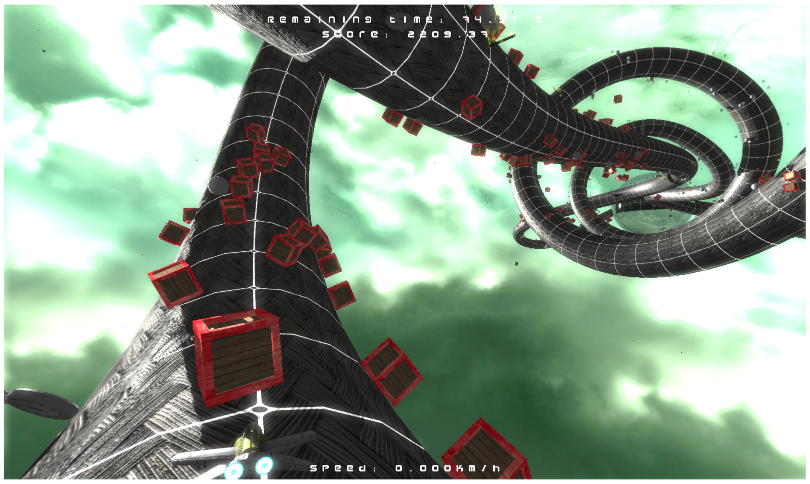

Figure 1: *Screenshot of the gameplay*

Abstract

TubeRacer is an interactive 3D action game. The player flies in a spaceship along a tube that runs through space. On the tube are various objects you can interact with (speed-boosts, items, obstacles). Goal of the game is to survive as long as possible.(cf. Figure 1)

Keywords: game programming, TubeRacer

1 Gameplay

In the beginning the player has a lifetime of 30 seconds. The lifetime decreases constantly, when it hits 0 the game is over. The player can collect clock items to increase his lifetime. However various obstacles are placed upon the tube like boxes and spheres. Hitting obstacles decreases lifetime. Furthermore along the track there are speedboosts which accelerate the spaceship for a limited period of time and also prevent the spaceship from colliding with the obstacles. The game is over when the lifetime reaches zero seconds. The player's score is measured by the distance he traveled. With each round the player completes the difficulty is increased, i.e. more obstacles are placed on the tube.

2 Graphic Effects

2.1 Lighting

For lighting we use a Phong lighting model with a global directional light source. In this lighting model, light is described as the sum of three different parts. An ambivalent part, which is background light, that has the same intensity everywhere. A diffuse part, which lights the surfaces directly facing the sun more than the ones diagonal to it. This part is calculated by comparing the surface normal with the light direction A specular part or reflection, depending on the viewers position.

2.2 Shadow Mapping

We implemented Shadow Mapping by rendering the depth buffer from the position of our sun into a texture, this way we would get the distance of the nearest object to the sun. By comparing the distance from every point, with the one from the depth buffer, we could decide whether an point would be seen by the sun or not (and thus be shadowed by something). However as we rendered the whole scene into only one picture, we would not get a high resolution of our shadow. To solve this problem, we rendered the near surroundings of the current camera-position into a second buffer This way we would have a high resolution for the shadow near the camera, and a lower one for the surrounding, where the resolution is not that important

2.3 Normal Mapping

We made massive use of normal mapping to give our game objects more structure. Normal mapping needs three vectors to define a vector space and thus a transformation of the normals saved in a texture to the normal of each fragment in world-space coordinates. The first vector is the normal of each vertex as saved in the VAO. We generate the second and third vector (called tangent and bitangent) in the geometry-shader from the UV-coordinates of each vertex. The normal-textures are generated in blender from the color-textures.

[*]jan.dreier@rwth-aachen.de
[†]denis.golovin@rwth-aachen.de
[‡]moritz.ibing@rwth-aachen.de
[§]simon.graetzer@rwth-aachen.de
[¶]jan.garcia@rwth-aachen.de

2.4 Bloom

This post-processing effect first applies a bright-pass filter upon the image. Then a bidirectional Gaussian blur is applied to the filtered image as well as two smaller versions of it. The three blurred images are stretched to the original size, merged together and then added to the original image. The images were resized before blurring to simulate big Gaussian-kernels in an efficient way.

2.5 Motion Blur

For implementing the motion blur, we compute the velocity for each vertex, by comparing its current position, with its position in previous frame. Then each fragment is blurred along its vector. We also apply a simple depth test to prevent objects in the back from blurring into objects before them.

2.6 Font Rendering

2.7 Particle System

Our particle system is run by emitters. Emitters have a lifetime and emit particles until they die. The trail behind the spaceship is generated by a long-living emitter whilst the emitter for each explosion only lives for a very short period. The emitter emits particles which are then passed to the shader-program. The geometry-shader turns each particle's position into a billboard facing the camera. The particles size and texture are derived from a type-attribute set by the emitter. The particle is drawn additive with alpha channel.

2.8 Environment Rendering

The glass-like orbs and reflective objects in the game are rendered using environment mapping. Environment mapping is done by first choosing an origin to render the reflections from, e.g. the center of a reflective object. Then we render the scene as seen from the origin in all directions: up, down, left, right, top, bottom.

3 Physics

Physics play a main role in the gameplay of our game. For example all the obstacles are attracted by the gravitation of the tube and interact physically correct with each other and the space ship. To do the physics simulation we used the bullet physics library. For the physics simulation to be more effective, we use simplified collision meshes of box-obstacles and the space ship. For the objects to fall towards the tube, we calculate the closest point on the tube and apply gravity in that direction.

4 Audio

Our audio support is facilitated by OpenAL. It is used to play menu music, the game music and ingame sound effects. We also use OpenAL to simulate 3D sounds when the space ship collides with an obstacle.

5 Images

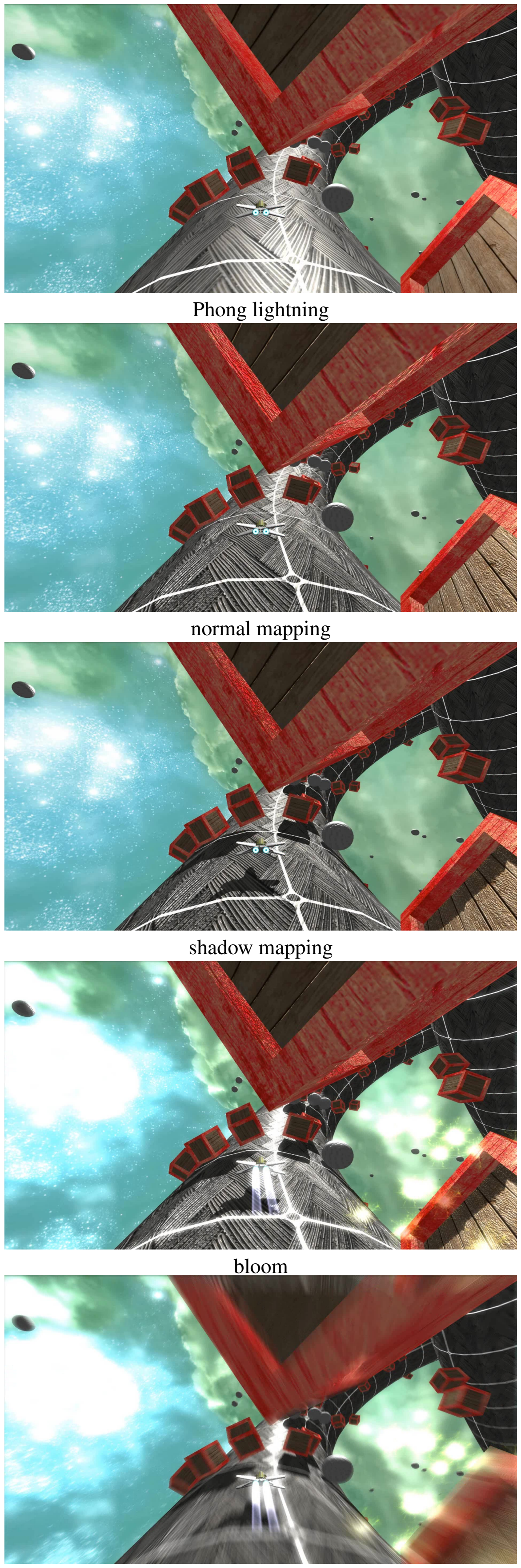

Figure 2: *Graphics effects used in the game.*