

# TetriMazeification

## Story

An insane king, which has a slight faible for board games, had his builders ordered to create a maze, to protect his unbelievable valuable treasure (just some gold, ...but he is proud of it ). To stand out from the many labyrinths of other crazy kings, a terrifying special effect was included to his project: The walls are moving!

A brave, ego-perspective discoverer and world champion in board games had the witty idea to get this fabled treasure of the crazy maze. The corridors in the maze where really dangerous because of some classical obstacles like arrow traps or hidden doors on the ground.

After a long desperate wandering in this desolate dungeon a magical fairy (with lots of cool graphic effects!) appears which our hero now follows for some reason.

The fairy leads our hero actually to the treasure. The discoverer nearly reached the old chest full of gold ..... but, oh no, he got squashed by a stone.

## Scene Description

Scene starts in the ego-perspective of the discoverer, following the fairy through the labyrinth. This scene is situated in the night under a beautiful sky full of stars.

As in the story described, there will be a few sudden actions implemented (moving walls, shooting arrows, etc.), where the camera has to slide back and avoid confrontation.

In the end there will be the place with the sought-after treasure, which our discoverer will never reach.

## Description of how the requirements were implemented

We render the demo with OpenGL 4.3 *Core Profile*.

### Lighting model:

For lighting a phong shader is applied to every fragment.

### Textures:

Textures are used on the walls and ground of the demo, as well as for the ground beneath the water meshes, on the falling rock (included as a trap) and the treasure box at the end of the path. On the walls as well as on the rock we used simple normal mapping to give the objects structure and a "surface".

### Fairy-, Camera- and Camera-View Path:

To make it easy to adjust as well as to implement, we used markers, made in blender, to describe different paths of different elements. The specified startpoint follows the imported markers one by one at a adjustable speed multiplied by our delta time.

To accomplish this, we created a pathwalker class, which evaluates and calculates the next direction and position, to get a smooth move.

This is used for the fairy as well as for the camera and also for the camera-view, which is nearly the same path as the camera, but slightly more ahead and sometimes is used to simulate the hero looking left, right for danger or smooth view-swing following the fairy with the "heros eyes".

### Camera:

The cameras trace is defined by an instance of a pathwalker. There are two camera modes which are switched during the rendering.

1. Mode: The camera looks into the direction of movement. This is implemented by setting the view target always just in front of the camera. When the direction is changed, the algorithm reacts on this by slowly moving the view target to the new direction to get a smooth rotation. This mode serves well for simple movements in a orthogonal grid.
2. Mode: A second pathwalker is used to serve as view direction. For more complicated camera moves, this technique was used, but for simple grid walking the other mode was more useful.

Additionally to this basic movement modes, different maneuvers are applied to fit to the respective situation,

1. While walking, the camera goes up and down and sways left and right to simulate movement by feet.
2. To dodge the spears, the camera goes down and waits their shortly.
3. The camera looks up to the stone which threatens to fall on it.

### Scene and its elements

- **Moving Walls:** Adjusted to the camera path, three walls are moving scripted with a specific delay.
- **Treasurebox:** In the end of the labyrinth is a chest full of treasures which in our storyline our hero wants to get.
- **Skybox:** Is implemented to give the demo a suiting ambience.
- **Traps:** A Falling stone and an arrow are implemented to create our desired scene and our hero tries to avoid getting hurt. The falling stone is visible in a scene nearly dropping onto the hero/camera. The arrow shots can be seen shortly after the beginning, when the camera is turning left.
- **Fairy:** The path of the fairy is easily adapted and can be modified with the path-markes in the code. (see below in How were the effects implemented for further information)

### Additional Resources and Libraries (including URLs)

In the current project there are several libraries included, some of these already existed in the used game of the lecture Computergraphics.

Included is the PhysX-Library, which is used for collision detection and the Assimp-Library to load the blender dae-files into the game. Also the freetype-Library is used to write 2D information onto the game-screen, this information can be loaded through different key-combinations. For the skybox we used the stb-Library to load images and some libraries where already included by the CG-Framework as there is, glut, glfw and glew. For implementing exciting audio we used the irrKlang-library to import and play the audio files.

**PhysX:**

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

**Assimp:**

<http://assimp.org/>

**freetype:**

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

**stb:**

<https://github.com/nothings/stb>

**glut:**

<https://www.opengl.org/resources/libraries/>

**glfw:**

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

**glew:**

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

**irrKlang:**

<https://www.ambiera.com/irrklang/>

## Which effects were implemented

For our final Version we included several effects. Two effects were implemented for the final grading, which are the particle-system on the GPU to model the fairy and the motion blur to give the demo a nice feeling.

Additionally some more effects were implemented or kept from the previous used game, to make the demo a little bit more interesting. These effects include simple-normal-mapping, water-mesh and a glow effect to put on the fairy in the fragment shader of the particles.

## How were the effects implemented

## Particle-System on GPU

Is used on the guidepost to let it seem more mystical and appropriate to the scene we want to create.

For this Particle-System we used a Compute-Shader and Geometry-Shader, as well as the usual Vertex- and Fragment-Shader. The Compute-Shader is used to control and emit particles at a given spawn-rate. Additionally here, we implemented a simplified version of perlin noise function called simplex to make the particle-system a more random and natural, as the velocity and position depend on it.

This noise-function were implemented with the help of several sources (see below).

To not only implement a random particle-system we added a vortex flow to control the movement in at least some way. To achieve this flow, we commit the center position of the vortex to the compute-shader and use some simple calculations to adjust the particles to this vortex.

Additionally spawned particles, again were manipulated with the given simplex noise-function to achieve spikes.

In the geometry-shader billboarding is implemented to alter the particles, which still are just points, into quads, which change according the camera-view to always face the camera.

Also implemented in the geometry-shader is the change of the particle-quad-size, if a given particle only has a remaining lifespan of 5 or lower. This lets the dying particles fade into smaller and smaller quads until the remaining lifespan is 0.

Vertex- and fragment-shaders are as usual, only the glow effect (see below) was included into the fragment-shader.

Information can be found on the following site:

### **Particle Effect/Systems:**

“Particle Systems with Compute & Geometry Shader” and “CGUE Content Particle Effects” by Institute of Computer Graphics and Algorithms

### **Particle-Flow, Simplex-Noise and Vortex**

[https://arm-software.github.io/opengl-es-sdk-for-android/compute\\_particles.html](https://arm-software.github.io/opengl-es-sdk-for-android/compute_particles.html)

<https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf>

<https://catlikecoding.com/unity/tutorials/simplex-noise/>

<http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>

[https://github.com/kbladin/Curl\\_Noise/blob/master/shaders/point\\_cloud\\_programs/update\\_velocities\\_curl\\_noise\\_vortex.frag](https://github.com/kbladin/Curl_Noise/blob/master/shaders/point_cloud_programs/update_velocities_curl_noise_vortex.frag) (Used as reference for the permutation in simplex noise)

<https://gamedevelopment.tutsplus.com/tutorials/adding-turbulence-to-a-particle-system--gamedev-13332>

## Motion Blur

For Motion Blur a post processing shader was implemented. For that, the scene is rendered into a framebuffer. This is then rendered as a texture on a plane in front of the camera. For this plane, the post processing shader is applied to be able to use screen space effects.

After rendering, the framebuffer is copied to a second one so that in every next iteration two successive frames are available.

The shader computes an average of two succeeding screens to enable a blur effect.

## Simple Normal Mapping

Is used to give the illusion of 3D wall texture and ground in the labyrinth. This effect was already implemented in the “Computergrafik” exercise project, called “Get Lost” by Alexandra Gamsjäger.

Information can be found on the following sites:

<https://www.pluralsight.com/blog/film-games/bump-normal-and-displacement-maps>

<http://ogldev.atspace.co.uk/www/tutorial26/tutorial26.html>

<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

and in the “Computergrafik”-Script “Texturing” by Peter Sikachev, Andreas H. König

## Glow Effect

Inspired by <https://www.shadertoy.com/view/4lfXRf>

**Glow Effect:** [https://developer.nvidia.com/gpugems/GPUGems/gpugems\\_ch21.html](https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch21.html)

and in the “Computergrafik”-Slides:

**Glow Effect:** “Per-Pixel Post Processing Bloom & Glow” by Simon Maximilian Fraiss

## What Tools have you used to create the Models

To create models as well as markers for the paths, we used blender as well as given models from open-sources.

## Resources:

The 3d model of the spear and of the rock origin from <https://www.turbosquid.com/>.

The other models were handcrafted by us.

## Controls

ESC: Stop the application/demo

F3: switching between wireframe mode on and off (in auto cam only the cubes inside, used for the camera position is shown)

F4: Prints out position and passed time of the automatic camera

F8: Switches Culling on and off

F9: Switching between fixed cam and free cam

A,S,D,W: free cam control only available in free cam mode

Space: Jump with free cam mode on

Mouse: adjust camera view/walk direction in free cam mode

**Tested Graphiccards:** NVIDIA GeForce GTX 560 Ti and NVIDIA GeForce GTX 1080

## Audio References:

<https://freesound.org/people/SoundFlakes/sounds/492244/>

<https://freesound.org/people/SoundFlakes/sounds/492535/>

<https://freesound.org/people/AGC66/sounds/393865/>

<https://freesound.org/people/AGC66/sounds/393865/>

<https://freesound.org/people/LittleRobotSoundFactory/sounds/270437/>

<https://freesound.org/people/jdupre01/sounds/484983/>

<https://freesound.org/people/TeamMasaka/sounds/197426/>

<https://freesound.org/people/JohnsonBrandEditing/sounds/244310/>

<https://freesound.org/people/newagesoup/sounds/377828/>