

Documentation - Monstar

1. Brief description

In Monstar the player tries to save little black monsters with bringing a shooting star back home. To do so, you have to push the fallen star through a maze, the goal is on the other end of the field.

2. Features

User Interaction	WASD + Mouse: Player can move monster ESC: Quits Game F1: Toggle wire-frame mode (implemented, at the moment, overdrawn by bloom implementation) F2: Toggle back-face culling (same as F1) F3: Toggle to Free-FPS Camera-Mode for Debugging/Inspecting the Scene [WASD + Mouse: to move the camera, Space and L-Shift for Up and Down Movements] F4: change to Idle Animation (is not shown correctly) F5: change to Running Animation (is not shown correctly)
3D Objects	Static: <ul style="list-style-type: none">- 3 Walls- Goal (has a wall shape and reflects the skybox)- Ramp/bridge Dynamic <ul style="list-style-type: none">- Player (black monster, animated, WASD interaction)- Shooting star (movable by player)- 2 Stone Enemies (You can knock them over)
Lightning	The scene is illuminated by the shooting star object. (Point Light)
Camera	The Camera follows the player, but can be moved with the mouse (as the feedback suggested). For debugging purposes, you can toggle to an FPS camera
Win/Loose Condition	Player wins if shooting star touches goal. The window closes and the console prints out a message. If the shooting star touches an enemy, the same happens.
Adjustable Parameter	Screen Resolution, Fullscreen-Mode, RefreshRate and brightness are read from a settings.ini file

Collision Detection - Basic Physics	The shooting star and the player are controlled by a physics engine (PhysX). Implemented in PhysXScenery.h and CustomPhysXCallbackHandler.h (plus corresponding .cpp files). It detects all collisions between them and other objects of the game and handles them correctly.
Advanced Physics	The shooting star moves correlating to the players' velocity when hit by the monster. Also player and shooting star are bouncing back from walls, etc. (Note: PhysX does something crazy on our AMD hardware, where it starts really fast, even though we set the max velocity, and after some time it slows down real slow)

3. Additional Libraries

GLEW, GLFW	For basic window and OpenGL functionalities. URL: http://glew.sourceforge.net/install.html (glew), https://www.glfw.org/ (glfw)
PhsX	Collision Handling and adding "real" physics effects like gravity URL: https://developer.nvidia.com/physx-sdk
assimp	Loading and handling animations URL https://www.assimp.org/index.php/downloads

4. Effects

Post-Processing: Bloom/Glow	The shooting star is glowing. Implemented in three shaders and the use- and initbloom function. Draws from framebuffer and uses Gauss filter and mipmap in last shader to make edges of bright objects blurry.
Animation GPU Vertex Skinning	(Note: we really(!) tried to make this effect work, but we couldn't get it to display the animation correctly. However, the player-character has animated legs, the rotation and translation get interpolated and the matrices for the jointTransforms are being weighted in the shader, but we couldn't get the formula right for the matrices given to the shader via the jointTransform Uniform). The player has moving legs. Implemented in an extra class structure using assimp. The model was animated using blender.

	The bone-skeleton, its weights and each Keyframe animation is imported. While rendering the model, the transformations according to the bones and its weights get interpolated between the timepoints of the keyframes.
Environment Map	We implemented a cubemap/skybox and created a new shader, which draws the reflections of the skybox on the surface according to the viewers (the cameras) angle. Only the goal has this shading. (Honestly, we implemented this, because we needed points, it doesn't fit the gameplay, but the skybox looks pretty).
Super Simple-Cell-Shading	Since we have a lot of flat Surfaces, it isn't visible too good, but we do have altered our phong-shading of most objects in the scene (the ground plane, the walls and the bridge), such that the ambient and diffuse factors are being drawn in levels (instead of a steady gradient). This gives simple a cell-shading look.

5. Step-by-step Guide

As already mentioned, the game is played by pushing the glowing shooting star to the other side of the maze. To facilitate things for test purposes, here a few important tips. First, the goal is straight ahead on the other side of the map. Sometimes moving the shooting star can be a bit tricky. But if you make your first push just hard enough, you can shoot the ball over the obstacle right in front of you. This would be the easiest solution, otherwise you have to make controlled movements to not push your star. And last, the map is surrounded by an invisible wall, just in case you don't notice it while playing.

You win if the shooting star touches the goal, which is the reflecting surface on the end of the map. Then the window automatically closes. If you pushed the star into an enemy (big stone guys), you lost and as before, the window closes. The output of the console shows a message according to your victory or defeat.

6. Additional Notes

6.1 Code used

function to convert assimp Mat4 to glm::mat4:

<https://stackoverflow.com/questions/29184311/how-to-rotate-a-skinned-models-bones-in-c-using-assimp>