

Greed

Nicolas Eder, David Köppl

June 13, 2022

1 What is Greed

Greed is a first person game written in C++ and OpenGL by Nicolas Eder and David Köppl.

1.1 Story

Two days of the dark caverns lay behind you, all for a treasure cave you weren't even sure exists. Now all doubt is gone - mountains of gold and valuables lay at your feet. How much of it will you be able to carry out though? And how long do you have until the "inactive" volcano you're standing in will bathe you in molten stone?

1.2 Gameplay Guide

The player starts the game in a collapsed mine shaft. A waypoint points to the center of a pit cave filled with treasure. There the player can collect golden items and get a feeling for moving around. Each collected item slows the movement down. The player can leave the cave through the bridge marked by a waypoint. Walking on this bridge triggers the lava to start rising. Now the player has to quickly reach the exit at the top of the cave. The player can choose different paths and solve some puzzles on the way up. If the exit is reached a score, based on the collected items and the time needed to reach the exit is displayed and the player can try again by pressing R. If the lava reaches the player, the game is over and needs to be restarted by also pressing R.

2 Game Features

2.1 Gameplay

- **3D Geometry** - multiple 3D Meshes, which were modeled by ourselves in Maya, get loaded from a .fbx file and converted into a GPU friendly data structure.
- **Playable** - The player can run around and collect items
- **60 FPS and Framerate Indipendence** - Physics and animations use a time delta and our game runs at about 100-240 fps on our machines. The frame rate is visible on the window title if full-screen is set to false in the settings.ini (the game runs windowed).
- **Advanced Gameplay** - The player has to jump over obstacles and solve puzzles while avoiding falling into lava
- **Win/Lose Condition** - The player wins if he reaches the top of the cave, and loses if he touches lava
- **Intuitive Camera and Controls** - to control the player use:
 - WASD - move player horizontally
 - Spacebar - jump up
 - Mouse - rotate camera
 - LMB - collect item

We used a quaternion based first person camera from the recipe "Working with a 3D camera and basic user interaction" in [SK21]. We have a free camera for debugging which can be activated by pressing F6 and is controlled by:

- WASD - move on cameras xz plane
- 1, 2 - move up/down along the cameras y axis
- LSHIFT - move faster
- Mouse - left click and drag rotates the camera

We also have an animated camera for recording the teaser, which can be used by pressing F10. Start and end position and orientation need to be set before compiling. More debug options and inputs are described in the readme.

- **Illumination Model** - we have a directional light (as the sun) and several point lights (visible as small torches) to illuminate the cave. Every mesh has a material consisting of up to 7 textures for our metallic-roughness PBR shader and normal vectors on every vertex.
- **Textures** - meshes can have the following textures: albedo, normal map, metallic, roughness, ambient occlusion, emissive and height. We generate mipmaps on load and use anisotropic filtering via the [GL_EXT_texture_filter_anisotropic](#) extension. The textures are created in Adobe Substance. The HDRI images are from [Poly Haven](#) and some textures are from [here](#). We use a blue noise texture from [here](#)
- **Movable Objects** - golden objects move based on the physics simulation. The lava plane also moves up after the player triggers it (walks on the first bridge).
- **Adjustable Parameters** - edit the settings.ini file located in /bin/assets/ to change some parameters:
 - width, height - sets the window resolution
 - fullscreen - sets the window to full-screen, assuming the set resolution matches the monitors' resolution
 - refresh - sets the refresh rate of the window
 - exposure - changes the scenes brightness, higher values correspond to a brighter image

To see what the other options do, you can look at the readme.

- **Collision Detection** - is implemented using the bullet library and keeps the player in bounds.
- **Advanced Physics** - at the top of our level we use bullet for some physics puzzles. If the player wants to collect an item we use the physics engine to ray-cast and pick the item if it is there.
- **View-Frustum Culling** - We generate an AABB for every mesh by finding the smallest and largest vertex, transforming them by their model matrix and selecting the smallest and largest transformed vertex. By pressing F8 you can toggle frustum culling which works by [extracting planes](#) and corners form the view-projection matrix. It tests against frustum planes and frustum boxes as described [here](#). If frustum culling is active, pressing F2 will print the number of culled objects to the console and draws the AABBs (green) and the cameras' frustum (yellow). Pressing then F7 will freeze the culling frustum in place making it possible to see the culling effect when moving the camera around.
- **Heads-Up Display** - Our HUD renders the text seen in the top left corner of the window, some icons in the bottom left corner, a crosshair in the center and a waypoint marker. It can be toggled with F9. The text gets rendered like in [this tutorial](#), the items are textures rendered to only a part of the viewport and the waypoint is a quad rendered as a billboard. The font used is [Quasimoda Regular](#) by lettersoup and is used under the free team license.

2.2 Effects

- **Lighting: Shadow Map with PCF** - Our directional (sun) light casts a shadow. We create an orthogonal projection using tight scene bounds and a view matrix from the sun's direction. Then we render the scene, without frustum culling, onto a depth texture with our light-view-projection matrix. In our main light shader we sample the depth texture with a PCF kernel of 7 to check if surrounding fragments are in shadow or not, as it was done in "Implementing shadow maps in OpenGL" in [SK21]. This gives us a shadow value between 0 and 1 which gets multiplied by the light contribution. The shadow without illumination and textures can be viewed by pressing F11. To avoid artifacts we use a bias value and strictly modeled closed meshes. The shadow map resolution can be changed in the settings.
- **Advanced Modeling: GPU Particle System using Compute Shader** - to simulate flowing lava we used the recipe "Implementing a particle simulation with the compute shader" in [Wol18] and modified it to be able to draw billboards instead of points. It works by setting up shader buffers for the positions and velocities of the particles, updating them every frame and resetting stray particles, then rendering an instanced quad with a model matrix constructed from the positions buffer. To get a glowing look the particles are rendered with blending set to add colors. The lava flow is visible after triggering the lava and is located to the left of the starting position.
- **Animation: Vertex Shader Animation** - Based on the recipe "Animating a surface with vertex displacement" in [Wol18] we animated the lava plane to look like it has waves and extended the wave effect in both x and z direction. This is visible from the start, when going close to the lava pool. The effect breaks if LOD is turned on, as the indices are out of order.
- **Texturing: Procedural Texture** - we took the method from the recipe "Creating a noise texture using GLM" in [Wol18] and extended it to the z axis to create 3D perlin noise, which gets sampled as an offset for the volumetric light. If you stand close and still you can see that the samples change over time giving the volume a cloud like texture. It is clearly visible after pressing F11. We also use image based lighting, but environment maps have been implemented in the ECG course.
- **Shading: Simple Normal Mapping** - Normal mapping is implemented with the help of [this article](#). The effect can be toggled by pressing F5. The difference should be immediately visible. We also have implemented a physically based shader for a metallic-roughness workflow, based on the code from [this Khronos project](#) and the recipe "Implementing the glTF2 shading model" from [SK21], but a different PBR shader was also implemented in the ECG course.
- **Advanced Data Structures: LOD using an octree** - we generate up to 8 different discrete level of details for every mesh we load, each level halving the number of indices. With the smallest LOD being 1024 indices. The methods are from the recipe "Generating LODs using MeshOptimizer" [SK21]. When testing we discovered the use of LODs doesn't change the performance much, as we don't have very detailed meshes and the LOD changes are very visible, so we decided to not use them. They can be enabled in the settings.
- **Post Processing: Bloom/Glow** - The luminance of the whole scene gets reduced to a single texel describing the average luminance of the scene. Using a compute shader the exponential light adaption from [this formula](#) [SNP00] gets calculated. Then bright pixels are extracted from the rendered scene and blurred with two ping pong textures. In the last step the original scene image gets combined with the blurred image. To tone map the colors a Reinhard 2 operator is applied to the image as it was done in the recipe "Implementing HDR rendering and tone mapping" [SK21]. You can look at the sky or at the lava particles and toggle the HDR-bloom pipeline with F3 to see the difference. Light adaption may not work on Intel GPUs, see bugs. Bloom strength can be changed in the settings.

2.3 Additional Features and Effects

- **Volumetric Light** - uses the same depth map as our shadow mapping and is implemented using the code from [this project](#) which was the winner of the real time graphics course in 2017. We

changed the dithering method by using blue noise, simplified the volume blurring and skipped the down/up sampling which introduced some bleeding but made it run faster. The ray-march step count can be changed in the settings.

- **3DLUT** - after tone mapping we apply a color grade see Chapter 24 in [Pha05] using the code from [this blog](#) to load 3DLUT files in .CUBE format.
- **SSAO** - we used the recipe "Implementing SSAO in OpenGL" in [SK21] to add a simple occlusion effect and generate our SSAO pattern as described in the chapter "Screen space ambient occlusion" in [Wol18]. It is clearly visible after pressing F11.
- **Audio Engine** - we implemented the observer pattern from [Nys11] to be able to easily play audio when some event happens. If the player starts moving we play footstep sounds, if the player triggers the lava, the music changes and so on. The sound effects and music were created in Ableton Live by David Köppl.
- **KTX Texture Compression** - we used to load .png textures with stb image, but using 7 textures per material makes loading times scale badly. Loading images that way also consumes way more memory than needed as [illustrated by Khronos](#) so we used the [toktx tool](#) to convert all .png files and [cmtf Studio](#) to convert the .hdr files to .ktx. We wanted to use ktx2 but this format doesn't have loaders for our 32 bit build.
- **GPU Data structures** - we started with a simple BVH tree as our scene graph. The bullet library updated the position and orientation of some selected objects and every frame the whole tree got traversed and for every model in every node a model matrix got calculated. This turned out to be really inefficient as most models don't move. Our solution was to make everything an array or vector to be precise. So our scene was an array of models, each model was a collection of indices to other arrays. Every loaded mesh was added to a big vertex and index array under a single VAO. The model struct then saves the starting index into the vertex and index arrays and the count of how many vertices and indices the model has. LODs are then just offset indices in memory and can be easily switched to. Our transformations are saved as a vector for position and a quaternion for orientation, scale was not needed. This saves us memory compared to a mat4 and was faster for the physics updates. We also made use of [bindless textures](#) to be able to draw the whole scene with just a single draw call using indirect command buffers, which cuts the time needed for switching textures completely and reduces synchronization time between CPU/GPU to a minimum. Our data structures make it possible to do the frustum culling, LOD selection and building the indirect render commands in parallel and directly on the GPU which would significantly speed up rendering for large scenes.

2.4 Bugs

- there seem to be memory leaks, so restarting the game (pressing R) often, may cause problems
- compression artifacts of the normal map may distort smooth surfaces
- images gets infinitely brighter: some Intel GPUs have problems using [texture views of frame-buffers](#) which breaks the light adaption. Disabling HDR (F3) somewhat fixes it. If such a GPU is detected the image brightness is capped and needs to be adjusted via the settings.
- weird textures: if a GPU doesn't support the extensions [GL_ARB.bindless_texture](#) or [GL_ARB.gpu_shader_int64](#) the scene gets drawn with fallback textures.

3 Libraries

- [assimp](#) - used to load models from a .fbx file.
- [bullet](#) - used for physics simulation
- [freetype](#) - is used to convert fonts to bitmaps for text rendering.

- [OpenGL Image \(GLI\)](#) - is used to load .ktx texture files
- [irrKlang](#) - used for playback of music and sound effects
- [MeshOptimizer](#) - used for indexing and simplifying geometry data and generating LOD meshes
- [Optick](#) - used for profiling the game, to get a profile dump run the debug build

References

- [Nys11] Robert Nystrom. *Game Programming Patterns*. APress, 2011.
- [Pha05] Matt Phar. *GPU Gems 2*. Addison Wesley, 2005.
- [SK21] Viktor Latypov Sergey Kosarevsky. *3D Graphics Rendering Cookbook*. Packt Publishing, 2021.
- [SNP00] Hector Yee Donald P. Greenberg Sumanta N. Pattanaik, Jack Tumblin. Time-dependent visual adaptation for fast realistic image display. 2000.
- [Wol18] David Wolff. *OpenGL 4 Shading Language Cookbook Third Edition*. Packt Publishing, 2018.