ROCKET AND PLANETS

UNIVERSITY OF BUCHAREST

COMPUTER GRAPHICS 3D PROJECT

PĂCURARIU RĂZVAN MIHAI LUPARU IOAN TEODOR

2025-2026

CONTENTS

CONCEPT

The scope of this project was to implement a base game engine fitted with a useful enough scene graph system, and on top of it build a simple demo game to showcase it and how it can be easily extended.

USED CONCEPTS AND ELEMENTS

This project uses batched rendering to minimize the amount of drawcalls issued, alongside frustum culling. Cascaded shadow mapping is used to allow shadows to have higher detail closer up. On the backend, things such as geometry shaders and Uniform Buffer Objects were used for some of the shader programs. Models can have a custom metallicity map, which changes how much a part of the model looks metallic.

DESCRIPTION AND ORIGINALITY

The project includes in its game scene a planet in the middle of the scene, alongside a slightly smaller moon thar orbits it, and also an asteroid ring further out with about a thousand asteroids. A lot further out there's a quad with a texture of a sun, in the direction from which the light of the scene is coming from. This gives the appearance of a real sun very far out, enhancing the feeling of a vast space.

On the 'side' of the planet there also spawns a rocket. This rocket is gravitationally attracted to both the planet and the moon, and it can also collide with them. It has a certain amount of fuel and charge, and if consumed the rocket can no longer fire its engine or stabilize. The 'goal' of the game could be to land the rocket on the smaller moon safely without landing on the side, too hard, or without consuming all its fuel and charge (so it could return back to the planet). The rocket being in 'stabilization' mode means that its rotation will 'stabilize', as in dampen, over time until it eventually stops.

On the screen, 3 GUI text elements are also present. One of them in the upper left corner just says the current FPS; the one in the bottom middle tells stats about the rocket, such as how much fuel and charge it has left, and also if its stabilization is turned on; lastly the one in the top middle says to which body('s surface) the rocket is closer to, the planet or the moon, and what that distance is. These GUI elements reposition themselves properly if the window is resized.

As for controls, the rocket has a multitude of them. First off, the rocket's engine can be turned on by holding `space`, which also turns on a visual particle emitter spewing out particles from the engine's nozzle. The ship's pitch, yaw and roll can also be controlled by holding `s`/`w`, `a`/`d` or `q`/`e` respectively. The stabilization mode of the rocket can be toggled by pressing `r`. Besides these, pressing 'b' turns on a debug view where the bounding boxes of renderables can be seen, which are used for frustum culling.

On the backend side of things, the engine has an elevated level of complexity, allowing to support a wide number of scenarios.

First off, there are the resource managers. They handle preloading and storing the assets of the engine/game properly, including both OpenGL/graphics specific ones such as shaders and UBOs; and more regular assets such as textures or 3D models. The shader manager does reflection on the shaders using functions such as glGetProgramiv to get information on the uniforms and UBOs used in them, allowing us to use a generic function to set the value of a uniform, checking being done at runtime whether that uniform actually exists and what specific OpenGL uniform setting function to actually call. UBOs are stored in std140 layout to allow for easy correspondence with C++ structs. A bug was discovered where vec3s are treated as a 16-byte object instead of a 12-byte one on some vendors, and as such all vec3s in uniforms were replaced with vec4, which was done with macros. To facilitate this, the GL_ARB_shading_language_include was used. Materials store specific uniform, UBO and texture values to send to a shader during rendering, and are stored as a json. During preloading, the json is parsed to match it with the reflection data of the shader it has.

Similarly, is done with a 3D model's material file, which also stores which material corresponds to which mesh of the model.

The way the renderer works is that each frame it is provided with a collection of Renderables (which are frustum culled before being stored for that particular frame), and once it is ready to render, it batches them to minimize drawcalls. There are 3 different sets of renderables: regular opaque ones, transparent ones, and GUI ones, which are sorted, batched, and rendered differently. Each renderable has a render key which is used in sorting which, alongside a state cache to track the current OpenGL state, minimize state changes. The model matrices are passed along as instance data for the batches; and for the GUI elements an additional field which stores a rectangle in UV space is passed to both allow using a texture as a spritesheet and to be able to batch GUI elements which use the same texture together (such as text which use the same font texture). Before the main render pass there is a shadow pass which determines the type of light in the scene (directional or point), and calls the appropriate shader with the appropriate UBO Shadow data. Cascaded shadow mapping with 6 shadow maps is used for a directional light, and a cubemap is used for a point light. To be able to render the shadows to all 6 textures in a single drawcall, geometry shaders are used to parallelize the work.

The part of the engine the user is intended to interact with, the scene graph, is modelled after an Entity Component System, using the entt library for this. The scene contains a hierarchical tree structure of 'game'/scene' entities, with a bunch of useful methods to add, reparent and get the children/descendants of a node/entity. The entities contain a bunch of components, some of them being more backend related, such as NameComponent (giving each entity a name), or RootComponent (used solely by the Root entity); while others are related to the usual game logic such as the TransformComponent which gives an entity a (local) model matrix in the scene space, the RenderComponent which allows for an entity to be rendered, the RigidBodyComponent which allows forces/impulses to be applied to the entity and the ColliderComponent whch gives a collision 'mesh' to the entity. These entities and components are of course associated with systems which manage and give logic to them. There is the RenderSystem which is the one that gets the renderables from the scene and submits them to the renderer each frame. The TransformSystem handles updating and retrieving both the local transforms and model matrices of entities, in an efficient manner using dirty flags and lazy updating and such. The PhysicsSystem handles updating the model matrices of entities with (angular) velocities, and the CollisonSystem handles collision detection and resolution. Currently only sphere – sphere and sphere- cylinder collision is supported. Regular classes also exist for entities, providing useful methods for working with them (such as getting the global position of a TransformEnitity, which requires calling a method in the TransformSystem to make sure the world model matrix is up to date).

This engine is designed to be extensible, as can be seen in the /demo directories. The way a user is intended to use the engine is by deriving a class from the Scene class, possibly defining their own custom entities derived from the Entity class or its engine derived classes, with their own custom components and systems; and then creating and using them in the OnCreate and OnUpdate methods of the scene. This can be seen in this project with the AsteroidRing and Rocket classes, and the code present in TestScene.cpp.

All the textures (other than the font texture) present in this application were hand made by the team members and were not taken from an asset site.

IMAGES

Figure 1: The main view of the application. The rocket on the main planet can be seen, alongside with the shadows of the moon being cast. The asteroids can be faintly seen in the background. All the three text elements mentioned can also be seen.

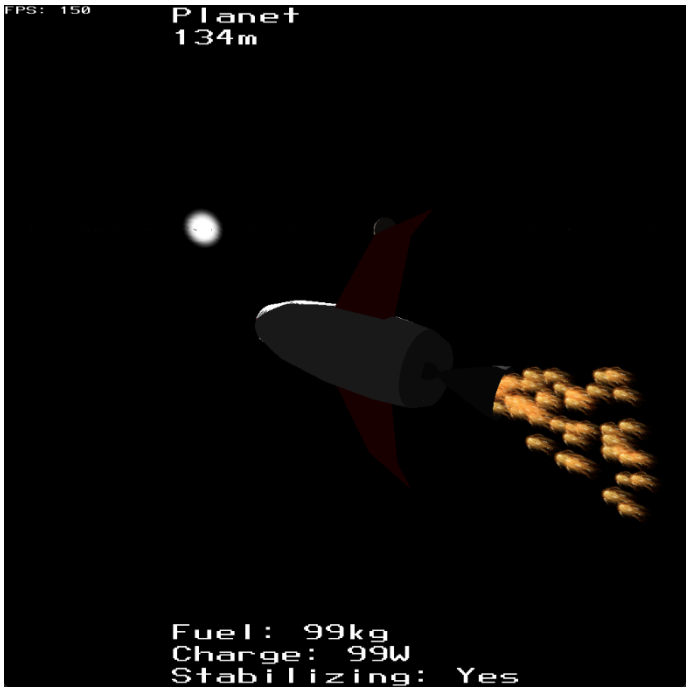

Figure 2: The rocket firing its engines towards a trajectory to the moon. The rocket's stabilization is turned on and some of its fuel and charge has been consumed.

Figure 3: The rocket after an unsuccessful landing on the moon. The collision between the rocket and the moon is illustrated here. The upper text has also updated to indicate that the moon is now the closest body to the rocket.

INDIVIDUAL CONTRIBUTIONS

Păcurariu Răzvan Mihai implemented the main engine.

Luparu Ioan Teodor implemented the test scene/game logic and also gave suggestions/resources for the engine implementation.

This project was more an illustration of the both the implementation, and the usage, of an engine, these two task being done by different people.

Both worked a roughly equal amount on the textures.

GITHUB

The repo for this project can be found at https://github.com/razvanpacku/ComputerGraphics-Project-3D.

REFERENCES

- This site was used as a reference for a bunch of the advanced concepts used in this project.
- The usual sources of help for fixing problems or figuring out how to do something (StackOverflow, LLMs, VS's autocomplete and copilot).
- The following external libraries were used:
  - assimp: used for loading 3D models
  - stb_image: used for loading textures
  - JSON for Modern C++: used for parsing the json material files
  - entt: used for ECS used by the scene graph system