ESAT
ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

> **Graphic Engine Programming**
> Arnau Rosselló
> Year 2023/2024

GRAPHIC ENGINE PROGRAMMING

ENGINE DOCUMENTATION

AUTHORS: Lázaro Piñol Palao
Ignacio Sempere Santonja

INDEX

1.- INTRODUCTION

This document exposes the different features of a graphic engine programmed using c++ and OpenGL. The engine offers core functionalities essential for graphic pipelines implementing techniques such as loading of meshes and textures, additive illumination with directional light, point lights and spot lights and its respective shadow mappings, forward rendering and deferred rendering.

The engine is based on an ECS (Entity Component System) where every object of the scene is represented by a single entity that links together one or several components. These components can be things like a renderer component, to render the object into the scene; a transform component that dotates objects with scale, rotation and location; a tree component, that allows to implement inheritance of values between the entity and its parent entity; a camera component, etc.

Moreover, the engine includes some other functionalities such as a scene graph to control every object of the scene, a flycam to easily navigate through the scene, a scene manager to easily store and load the different scenes created, audio management to reproduce different audio files, a job system that allows to throw tasks to different threads. Also the engine includes several menus and windows made with ImGui to easily access and control the different resources loaded on the engine.

Finally, a port to DirectX 11 and UWP of this engine has been made in order to fully load an scene in a XBox One dev kit and control the user input through an XBox controller.

2.- IMPLEMENTED TECHNIQUES

2.1 Entity Component System

2.1.1 Concept

An Entity Component System is a concept of programming applied to game engines and other programming environments, that revolves around the idea of splitting an object into its components, in order to manage each of them separately in big quantities quickly thanks to the components placement in memory..

An enemy could be splitted into the components of AI, Renderer, Physics, and others, and this could allow for each enemy to be moved at the same time and at higher speeds than if all its components were located together as one object.

2.1.2 Implementation

We have a “component manager” class, that in itself holds all the information of the components, each component belongs to an “entity”, a code generated before the creation of any component that links all the components belonging together. Each component is located in its own vector, next to others of the same time.

The component manager has a number of functions that help the management and update of the components, called “systems”, such as scene graph updating, for example.

2.1.3 Possible improvements

We could add more components that bring more functionality to the systems, such as Collisions, Frustum Culling, among others.

2.2 Scene graph and transformation inheritances

2.2.1 Concept

A scene graph is the concept of a visual representation, in the form of a list that contains other lists, of how one or many entities depend on others and can inherit some functionalities of the parents.

An entity that inherits the transformation of another entity, can be moved easily by the scene without having to even be updated itself at all, like how an object of a room would move only by moving the room itself.

2.2.2 Implementation

In order to build the graph scene, we identify and recollect all the entities that don’t inherit from another entity.
Once collected, we look at each of those and check their children, and recursively check the subsequent children of each of them, and build a list for each finding that holds all the children that belong to the searched entity.

2.2.3 Possible improvements

Faster iteration could maybe be achieved, and at the moment only the transformations inheritance is benefited from the graph scene, and the functionality could be expanded to other components and systems.

2.3 Scene management and database saving/loading

2.3.1 Concept

In order to manage, save and load scenes, we implemented a system class that manages how to save all the components and resources that create a scene into a database, that can be loaded later, without having to create a different file or alter the code itself.

2.3.2 Implementation

We used the SQLite library in order to create the database that holds all the information, which benefits from being faster to read than a text file, can be updated with the proper tools, and it's easier to read than a whole text file.

The system, when saving, iterates through all the components and resources, and turns them into SQL registers of the database.

When loading, it reads each line, and reconstructs the scene if all the textures and meshes still exist.

2.3.3 Possible improvements

Better and more accessible management could be made by adding more menus, a functionality to convert older databases into newer ones.

2.4 Cubemap

2.4.1 Concept

To enhance the feeling of the environment and give the sensation of an infinite playground we have implemented a cubemap, a cube mesh with inverted culling that renders a desired texture of a landscape.

2.4.2 Implementation

In the constructor of the engine we load a cube mesh and bind it a texture. We also set all the normals of the cube to 0 so in the render step, if a mesh normal is 0 the cubemap is drawn. To give the sensation of infiniteness the camera moves along with the cube.

2.4.3 Possible improvements

The cubemap uses a single orthogonal texture to render the landscape. An improvement could be rendering six independent textures to the cube (left, right, top, bottom, front, back) to give users more control on choosing how to draw the cubemap.

2.5 Text Rendering

2.5.1 Concept

In order to generate texts on the screen, like scores on a game or the controls of the camera, we implemented text rendering with the library FreeType.

2.5.2 Implementation

At the start, each letter from a font is loaded into a map of structs that contains the texture of that letter. When processing a text, each letter is rendered according to the character it represents, with a given font size and starting from a given position.

2.5.3 Possible improvements

The actual rendering of each character is costly and could be easily improved by having all the letters on the same texture, and only rendering into a single image, saving both time and effort in CPU and GPU terms.

2.6 Audio management

2.6.1 Concept

To manage different audio files on the engine we have integrated the OpenAL sound library to the engine. The implementation allows to play, pause, stop, change the pitch, change the volume and change the position of the different audio files.

2.6.2 Implementation

On the constructor of the engine we create the device and the context where the audio sources are going to reproduce. Then, with the help of ImGui windows we allow the user to control the audios that are loaded on the demo.

2.6.3 Possible improvements

The engine only allows loading WAV format files. A good improvement could be extending the format that the engine can load.

2.7 Directional, Spot and Point Lights

2.7.1 Concept

Each type of light brings different forms of illumination to the scene, and each of them has its own difficulties.
A directional light works like the sun, illuminating a scene from all directions equally, a spot light works like a streetlight, illuminating a small space, and a pointlight works like a bulb, illuminating the surrounding close space.

2.7.2 Implementation

For each type of light, a different shader and internal logic is followed, which then is applied to each element of the scene, and all of those are blended together to form a multiple lighted scene, but at a great cost for performance.

2.7.3 Possible improvements

The actual lights are a bit limited by aspects of design, and could be improved by multiple lighting techniques and design patterns, to reduce its impact on performance and give better illumination to the scenes.

2.8 Shadow mapping

2.8.1 Concept

Shadow mapping is the concept of using the scene and the position of lights to calculate and render where the objects should be illuminated or not.

2.8.2 Implementation

By using a concept called “depth map”, we can calculate which pixel, from the point of view of the said light, is closest to the camera. Then, after mapping the scene, we cross the data with the positions that the light will lit, and create shadows on the elements that would be obscured by an object.

2.8.3 Possible improvements

Dynamic maps could be implemented to automatically adjust to the surroundings of the light, and reduce the amount of configuration and setting that each light needs.

2.9 Forward rendering

2.9.1 Concept

Forward rendering is the concept of rendering each object of the scene once per light that affects it. It's very demanding on scenes with lots of objects rendered at the same time, but it's easy to implement, and can be better than deferred when a scene has few lights and few objects, which saves on memory costs.

2.9.2 Implementation

It's pretty straightforward, you render an object once for each light of any type that it's on the scene. In combination with additive lights and shadow mapping, it can create beautiful scenes but its very heavy on CPU performance and GPU calculations due to the high number of GPU calls that are needed.

2.9.3 Possible improvements

A system that determines when is it better to use forward rendering or deferred could be a great improvement, by calculating the costs of performance that would take each situation and applying the optimal one, along with other rendering techniques like frustum culling or lightning zones.

2.10 Deferred rendering

2.10.1 Concept

Deferred rendering works similar to Forward rendering, but with a twist: The rendering of geometries, textures, normals and other massive information, is only done once. And it is done and saved into different textures, so it can be easily read and accessed. This comes as a big memory cost, since storing multiple textures the same size as our window, can be very memory intensive.

2.10.2 Implementation

In order to implement Deferred Rendering, you first need to be familiar with the concept of a FrameBuffer, which is the place you render your elements into and then load it into the displaying window.

Our Deferred Rendering renders all the information, not on a framebuffer, but on separated textures, that we later use to calculate lights (only once per light), and render that into the final framebuffer.

2.10.3 Possible improvements

As with Forward Rendering, a system that determines which system adjusts better to the moment, and other rendering techniques, could greatly improve the performance and allow for bigger, beautier, and more efficient scenes.

2.11 Additive lights

2.11.1 Concept

The concept of making the lights additive implies having more than one type of illumination at the same time on the same scene to recreate better scenes. This also implies mixing all the types of shadows that cast each of the lights.

2.11.2 Implementation

In order to add more than one type of light and mix their colours and shadows to the respective meshes they affect we have blended them all. First we draw all the directional lights, then we blend additively the point lights and finally we blend additively the spot lights of the scene. We also have added controls on ImGui to decide if an object receives light or casts shadows.

2.11.3 Possible improvements

Some possible improvements of the additive lightning could be using less and better shaders to recreate this effect. Also some shadows that cast some objects are not perfect in the way that they are a bit pixelated or not well defined. Fixing this error could also improve the lightning system.

2.12 Job system

2.12.1 Concept

A job system is the concept of profiting the capabilities of multi-threading to split the costs of calculation, or reading information from the physical drives, in order to gain time and efficiency.

2.12.2 Implementation

By implementing a system that keeps a list of pending tasks, that a thread will receive and process when notified, we can free the main thread from having to stop and do heavy chores before continuing.

This, as usual, comes with risks, such as data races, which is the moment that two threads try to change the same variable in memory, which could lead to undefined behaviours and errors.

2.12.3 Possible improvements

By implementing a better job system, we could have multiple types of tasks that would split complete systems apart, such as rendering the whole scene, dealing with heavy calculations like transforms and physics, and such, like engines as Unreal do.

2.13 Obj loading

2.13.1 Concept

Obj loading is the concept of loading into memory big models to render, based on vertices, normals, and other elements, from files that need to be processed.

2.13.2 Implementation

By using the library TinyObj, which translates the original disk file into manageable information, and a bit of processing done by code, we can create big vectors of data, that later we load into buffers for the graphic library used to render the scene.

2.13.3 Possible improvements

At the moment, objects with multiple shapes are not loaded properly, and materials are also not loaded. In a future implementation, allowing for these elements to be processed correctly, we could bring more complex elements to the scenes.

2.14 Texture loading

2.14.1 Concept

Similar to how we did with Objects, we use a library, the STB_library in this case, to recover information from disks, and turn it into processable data. In this case, textures, which we use to paint better objects,with highly defined colours.

2.14.2 Implementation

The implementation is quite simple, just by calling the proper functions from the STB lib, we load textures from disk, and as they are multiple texture formats, we can load different types of textures, depending on the needs of the scene. Then, depending on the graphic library, load that information into buffers that the rendering system can use.

2.14.3 Possible improvements

By using the job system, we could dynamically create a loading queue of textures, and that could drastically improve the speed at which an application launches.

2.15 ImGui management

2.15.1 Concept

To ease the manipulation of light, meshes, textures and resources in general we have decided to implement several ImGui tools on the engine. These are grouped in a menu bar on the top of the window of each scene.

2.15.2 Implementation

We have added the ImGui library to the engine within the Conan file. Once the library was operative we designed the necessary tools and the better UI possible to give the user access to control almost everything graphically.

2.15.3 Possible improvements

There are some tools that could enhance the user experience as loading resources directly from the file explorer and not having to load all of them previously on the engine.

2.16 Flycam

2.16.1 Concept

The flycam allows the user to freely navigate through the scene, watching it from several angles and easing the design of the level.

2.16.2 Implementation

By storing the keyboard and the mouse input we allow the user to freely move the camera around the scene. This is also made by recalculating the projection and the view matrix of the camera every time that an input is stored.

2.16.3 Possible improvements

Implementing a shortcut to focus on different objects of the scene, rendering more than one camera at the same time or storing different locations and rotations of the camera to rapidly change among view could have been a good improvement for the flycam.

2.17 Gamepad support

2.17.1 Concept

In order to add gamepad support to the program, which adds depth to the possibilities of the applications implementation, we needed to map the inputs of a gamepad into the engine.

2.17.2 Implementation

Thanks to the XInput library from Windows, we can retrieve which buttons, triggers or joysticks are pressed at which moment, and thanks to a bit of logic, even have smooth movement depending on how much a joystick is moved or a trigger is pressed, something not possible with a keyboard, that only has pressed or unpressed states.

2.17.3 Possible improvements

Identifying and mapping multiple gamepads could allow for local co-op of games to be easily designed within the engine.

2.18 DirectX port and UWP port

2.18.1 Concept

In order to be able to port the engine into other platforms, such as Xbox, we needed to adjust the functionality of our engine to what that platform understands, and this can be a lot of work, even if rightfully planned ahead.

2.18.2 Implementation

Due to Xbox and UWP only working with DirectX, the engine had to adapt its rendering system to work with DirectX too. This meant reworking the obj loading, the textures, shaders and other elements almost entirely.

Things like creating a window with GLFW had to be recreated again, with other code, in order to be exported without errors. Others, like shaders, implied a complete different shading language.

The process, while really good to learn new architectures and gain multi platform experience, can take a great toll on the available time.

2.18.3 Possible improvements

The actual state of the port, while functional, it's lacking an amount of the elements of the original engine, such as multiple blending lights, shadows, and text rendering.

While it would be great to have parity between ports, it's an improvement that is not easy to achieve.

3.- TASK DISTRIBUTION

Technique	Tasks
ECS	Lázaro: Research and implementation Ignacio: Research and implementation
Scene Graph and Inheritance	Lázaro: Research and implementation
Scene Management and Database	Lázaro: Research and implementation
Cubemap	Ignacio: Research and implementation
Text Rendering	Lázaro: Research and implementation
Audio Management	Ignacio: Research and implementation
Directional, Spot and Point Light	Lázaro: Research and implementation Ignacio: Research and implementation
Shadow Mapping	Lázaro: Research and implementation Ignacio: Research
Forward Rendering	Lázaro: Research and implementation Ignacio: Research and implementation
Deferred Rendering	Lázaro: Research and implementation Ignacio: Research and implementation
Additive Lights	Lázaro: Research and implementation Ignacio: Research and implementation
Job System	Lázaro: Research and implementation Ignacio: Research an implementation
Obj Loading	Lázaro: Research and implementation Ignacio: Research an implementation
Texture Loading	Lázaro: Research and implementation Ignacio: Research an implementation
ImGui Management	Lázaro: Research and implementation Ignacio: Research an implementation
Flycam	Lázaro: Research and implementation Ignacio: Research
Gamepad Support	Lázaro: Research and implementation
DirectX and UWP port	Lázaro: Research and implementation Ignacio: Research

4.- IMPORTANT INFORMATION

SQLite Home Page, https://www.sqlite.org/index.html

Microsoft, https://learn.microsoft.com/es-es/windows/uwp/gaming/directx-getting-started

Bainbridge, Joshua. “tinyobjloader/tinyobjloader: Tiny but powerful single file wavefront obj loader.” *GitHub*, https://github.com/tinyobjloader/tinyobjloader
Chlumský, Viktor. “LearnOpenGL - Text Rendering.” *Learn OpenGL*, https://learnopengl.com/In-Practice/Text-Rendering

“LearnOpenGL - Cubemaps.” *Learn OpenGL*, https://learnopengl.com/Advanced-OpenGL/Cubemaps

“LearnOpenGL - Deferred Shading.” *Learn OpenGL*, https://learnopengl.com/Advanced-Lighting/Deferred-Shading

“LearnOpenGL - Light casters.” *Learn OpenGL*, https://learnopengl.com/Lighting/Light-casters

“LearnOpenGL - Scene Graph.” *Learn OpenGL*, https://learnopengl.com/Guest-Articles/2021/Scene/Scene-Graph

“LearnOpenGL - Shadow Mapping.” *Learn OpenGL*, https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

“nothings/stb: stb single-file public domain libraries for C/C++.” *GitHub*, https://github.com/nothings/stb

“ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies.” *GitHub*, https://github.com/ocornut/imgui