VORPAL ENGINE:

https://github.com/Aimar-Goni/3PMV_gonyibe_corderobe.git

Carlos Cordero Bernabeu
Aimar Goñi Benito

Engine Features

-Job system

A jobs system in a graphics engine is a way to automatically handle all the procedures associated with multithreading functionality in C++.

The system automatically manages thread distribution, locking the thread generation and usage until a new one is available, and organizes the incoming functions in a queue. These functions can be formatted with any parameters and a return value, which is returned afterward as a future of said value, from which we can get all the information needed to correctly multithread an application.

-ECS

An entity component system is a software architectural pattern commonly used in videogame development, its primary goal is storing all the information about anything that can be considered an entity (grouping mutable data that represent a specific object in the world) in the most cache-friendly way possible. This means that the memory is laid out in “components” in a continuous way, and the access of said data is also done sequentially in a manner where each component pack is traversed while the maximum amount of blocks of those arrays are in cache memory.

In our engine, we implemented an ECS that stores generic structs using the C++ templating system, allowing us to introduce new types of “components” to the system at runtime. Each entity is only stored as an integer which directs us to the location of said component pack in every array of components.

To maintain the components as tightly packed as possible, when deleting an entity and then adding a new one, the spot where the previously deleted entity gets priority as a place to lay out its memory as opposed to inserting the component at the end of the components array.

-ResourceManager

Our resource manager consists of a system that loads assets into the game engine and also stores references to said assets for later use or reference if needed.

Formats supported:
-obj Models, with their associated .mtl files
-images as Textures
-glsl shader code , vertex, fragment, and geometry shaders

The system used is designed to be as intuitive as possible for the user, with an `add` function that takes a shared pointer to the resource and its relative path to the data folder (which is also configurable in the global_config.h file). After adding every resource needed, the function `LoadAddedResources` is called, which does as it says. It's important to note that

only the models are loaded in a multithreaded way, which are coincidentally the most expensive types of resource to load. A `Get` function that takes the name of a resource is also provided as a way to access pointers to loaded resources.

-Camera

The camera is divided into two main parts: the editor camera and the game camera.

Editor camera: This camera is the one that is used while the user is in the editor. This camera has Flycam functionality so it allows the user to move freely around the scene. To move around, the user uses the AWSD and rotates the camera by holding the left click. There is also the possibility to change the speed of the camera by holding the left shit and turning the mouse wheel.

Game Camera: The game camera is the same as the editor camera but the user has to program it to do what he wants using the scripting function.

-Billboards

The billboard system is part of the UI program for the ease of use of the engine. Billboards are planes that always face the camera and indicate the position of invisible components in the editor, such as lights or cameras.

Firstly, the `InitBillboardManager` function initializes the billboard manager by creating an instance of `ComponentManager`, which is responsible for managing various components of entities. During this initialization, several component classes such as `BillboardComponent`, `TransformComponent`, `ModelComponent`, and `MaterialComponent` are registered with the `ComponentManager`. Additionally, it loads textures from the `ResourceManager` and associates them with specific materials.

Subsequently, the `GenerateBasicBillboard` function is used to create billboards. It begins by creating a new entity and assigning it a `BillboardComponent`, which defines the type (camera, light, or unknown), position, and visibility of the billboard. Following this, a `TransformComponent` is assigned to manage the position, rotation, and scale of the billboard. Depending on the type of billboard, a `MaterialComponent` is then assigned with the corresponding material (light or camera). Furthermore, a `PropertiesComponent` is added to give the entity a name, and a `ModelComponent` is set to define the geometric model of the billboard.

Moreover, the `UpdateBillboard` function periodically updates the billboards by iterating through all `BillboardComponent` instances and adjusting their transformations based on the camera's position. This involves calculating the direction from the billboard to the camera and determining the horizontal and vertical rotation angles to ensure the billboard always faces the camera.

-UI & Scene Hirearchy

The UI has been developed using the ImGui library. It has been one of the main priorities of this engine due to the ease of use it provides when using the engine.It has been developed with several functionalities in mind to make the user experience as easy and comfortable as possible.

-Scene hierarchy:
Here, using the tools offered by ImGui, a list of all the entities in the current scene is shown, along with the cameras and lights. It also allows you to select any entity.

-Data modification:
Another of the most useful functionalities is to be able to modify the entity that has been selected in the scene hierarchy. Here, depending on what you have selected in the hierarchy, you will be shown variables that can be modified. This ranges from the position of an entity to the color of the light.
Camera: Direction and position.
Entity: Position, rotation, scale, model, and material.
Light: On, shadows, position, color, ambient, diffusion, specular, specular, quadratic, constant, linear, direction and cut point.

In this window, the functions delete, rename, duplicate, and focus are also added.
Delete: Deletes the selected entity.
Rename: Allows to change the name that appears in the scene hierarchy.
Duplicate: Creates a copy with the same settings as the selected entity.
Focus: Moves the camera to the position of the entity, so it is easier to find objects.

-Add new components:
Another very useful feature of the UI is the option to create new entities and lights. To do this, just click on the add button and select what you want to add. The new entity will appear in the scene hierarchy.

-Various options:
In addition to everything else, the UI also allows you to enable and disable ambient occlusion, blur and modify values of these.

All these functionalities reinforce each other, because of this, new entities or changes made will be saved in the scene and can be accessed even after the engine is closed. The UI also offers the possibility to rearrange all the windows it uses so that each user can set his working area as comfortably as possible.

-Scene Saving and loading

One of the main features present in the engine is Scene saving and loading.

The user has the option to load and save scenes. In this way, all entity transforms, models, textures, lights, and cameras that are modified in execution can be saved for easy access later.

The chosen format to save the scene is JSON , as it is human-readable, easy to parse (which allows for external tooling to be developed), and easy to modify by hand in a pinch.

Saving is done with a JSON parser that writes arrays containing the data to be saved into a file: positions, rotations, directions... that can then do the reverse function to reload all the saved data. This tool allows you to load, save, and save in a new scene. In addition, the engine saves the last scene that has been modified and opens it when the engine is run again. Not only that but when exiting the engine, it launches a warning in case the user does not remember to save.

-PseudoScripting & Playmode

The main objective of a scripting subsystem attached to a game engine is to make gameplay programming easier, faster, and more intuitive, as well as isolating the internal engine subsystems and processes that we don't want to show the final user as either a way to protect internal functionality display or just as a way to leave a clean interface for the user.

In our case, we don't entirely use a scripting system as those usually involve a more simple language like Lua or Python being embedded into the engine. In our case, we apply the same principles but we keep the subsystem C++. In a way, we are aiming to replicate the Unity MonoBehaviour philosophy giving the user a minimalistic and simple file with access to most of the engine gameplay functionality.

How it works:

Externally to the engine files, EngineCore, alongside each example main file (for example, main_portals.cpp) , there lives a "gameplay" folder with an include and src directory. Inside those, you can find gameplay scripts.

These gameplay scripts inherit from a GameplayCore class which has access to typical gameplay functions such as
-Awake
-Start
-Update
-Cleanup

The user can overload these functions and make use of their parameters (ecs, input, timer, resource manager, gameplay camera, light manager) to make gameplay. The keep aspect that allows this system to function is that PLAYMODE accompanies this system: when you press the [PLAY] button at the top center of the UI, the functions Awake and Start are called once, Update is called every frame and Cleanup is called when [STOP] is pressed. Every system modified during PLAYMODE goes back to its initial state after it ends. This means pressing play many times allows you to test your gameplay functionality without modifying

the scene permanently. You also get a second gameplay camera, which you possess during play mode (the user loses access to the flycam during gameplay).

Many gameplay scripts can be created (multiple .h and .cpp files in the gameplay folder), but those have to be registered including themselves in the *gameplay_custom_modules.h* file, which can be done manually or using the tool at *tools/new_gameplay_script.bat*

-Portals

The rendering techniques used will be explained in the Techniques section.
We implemented portals as the main distinguishing feature of the engine. These are inspired by CodeParade's Non-Euclidean Worlds Engine Video.

Portals teleport seamlessly the user to another linked portal and render whatever the other portal sees if the player were to be on the other side of the portal. Portals can be linked by face, which means one portal can be its front and back linked with two different portals.

These portals make use of a minimalistic physics system to simulate portal-player collision. Portal recursion is allowed, which means portals can render other portals in a loop, which has a hard coded limit to the number of bounces it can perform.

Portals only support forward rendering, but lighting and shadows completely work.

-VR

The rendering techniques used will be explained in the Techniques section.
Virtual Reality is supported via OpenVR (VR SDK made by Valve, which is powered by the SteamVR runtime).

The VR functionality supported is complete access to the OpenVR SDK and its subsystems: input detection, HMD (Head Mount Display) positioning and variables, haptic feedback, VR boundary detection, and manipulation. Only HMD positioning and input detection are used in the demo scene.

When using the VR functionality, the user has access to the engine window on the PC side, which displays the two scenes being rendered to each eye of the VR headset in a split-screen manner. This is known as a "companion window" setup.

The setup works in a way where the only changes that need to be applied to our default Engine main file are:
-rerouting the Engine creation to be handled by VRInit
-assigning the main window to be displayed in the PC side to be the companion window
-calling VRRenderFrame instead of any other rendering function
-calling VRShutdown on closeup

The engine supports loading any scene with VR functionality, but portals and VR cannot be used together as of the current version of the Engine. It also doesn't display shadows and has limited access to the Blinn-Phong setup.

Setup needed: The tested setup is a Meta Quest 2 device, running through SteamVR with Meta Quest Cable Link enabled (device connected to a PC via cable), and after all that running the designated Visual Studio project (in our case named OPENVR).

-Input

This system handles keyboard and mouse inputs through dedicated classes, enabling the detection and processing of various input events, such as key presses and mouse movements.

The `Input` class is the main entry point for the input system. It initializes instances of `KeyInput` and `MouseInput` classes and sets up the necessary callbacks for GLFW to handle input events. The constructor of the `Input` class takes a `Window` object and sets up key and mouse input handling by associating them with the window. When `RENDERING_BACKEND_OPENGL` is defined, the GLFW window user pointer is set to this `Input` instance to allow the GLFW callbacks to access the input system.

The `ProcessInput` function polls GLFW events, ensuring that the application processes any pending input events from the user.

The `KeyInput` class is responsible for managing keyboard input. It maintains a list of monitored keys, initializes their states to false, and provides methods to check the state of a key (whether it is pressed, released, or held down). The `setupKeyInputs` method sets the GLFW key callback to handle key events. When a key event occurs, the callback updates the state of the key and records keys pressed or released during the current frame. The `updateKeyStates` method clears the lists of keys pressed or released in the current frame and updates the previous key states to the current ones.

The `MouseInput` class handles mouse input. It monitors a list of mouse buttons, initializes their states, and provides methods to check if a button is pressed. The `setupButtonInputs` method sets the GLFW scroll and mouse button callbacks to handle mouse events. The scroll callback handles vertical scroll events, updating the state of the corresponding buttons (8 for scroll down and 9 for scroll up). The mouse button callback updates the state of the mouse buttons. The class also includes methods to get the current mouse position and to adjust the mouse sensitivity.

Implemented Rendering Techniques:

-Blinn–Phong Lighting

Blinn-Phong lighting is a lighting model used to approximate the way reflections work when dealing with point, spot, and directional lights in a graphics engine environment.

The basic setup allows the use of reflections involving details generated by ambient, specular, and diffuse parameters associated with the lights and the objects they affect.

In our particular setup, the lights hold all the diffuse and specular data, with the only information provided by the affected objects being their mesh normals, vertex positions, and color. This is true for both the forward and deferred rendering pipelines implemented.

-Shadow mapping

Shadow mapping is a technique that allows for shadows to be added to a 3D graphics scene. The process involves drawing a shadow for everything each light can see from its point of view. This process works in the following way:

-A depth map from the point of view of the light is created(a greyscale render of a 3D scene where the pixels closest to the camera are rendered in black and go towards white the further away they get from it).
-A shadow map is generated using the depth map, which is simply a representation in black and white of the shape of the shadow inside a texture.
-Said shadow map is used alongside a matrix that stores the light space of the light and the view it was rendered to convert it to world space to determine where the shadow would be.

In the case of the pointlight, the scene has to be rendered from six different points of view to accurately represent its shadow, so a cubemap is used instead to store the shadow map.

-Combining Shadow Mapping and Blinn-Phong

For this setup to work, the lighting of the scene has to be computed once for each light alongside its lightspace matrix and shadow map to create its shadow. We went with two different approaches as time went on:

Prior to Portals
The scene was rendered once for every light, this approach worked and had the added benefit that we could add as many lights as we wanted to the scene, but adding multiple lights made the scene rendering extremely expensive.

After Portals:
After we added portal functionality, the scene ran extremely slow (on the order of 10fps on average). The cause of the slowdown was that, the whole pipeline had to be run for each portal reflection. In order to be able to use portals comfortably on a real scene, we had to change the way forward rendering worked, by hard coding the number of lights that were in the scene, and rendering the scene only once computing every light from an array of lights.

For this to work we had to move to texture arrays for shadow maps, which allow for any number of textures to be stored in a 3D texture, and keep an array of textures (opengl only allows for 16 or 32 textures this way, which have to be active all at once) for the cubemap of the point light. Each light also computes its own lightspace fragment position sending its own shadow map and lightspace matrix in the form of a struct.

Light uniform sending was also moved to be done using Uniform Buffer Objects, having to line up memory layouts to those of OpenGL stipulations by introducing padding in the light structure on the C++ side.

-Forward Rendering

Forward rendering is the standard technique that most engines use for rendering scenes. You supply the graphics card a geometry, it projects it and breaks it down into vertices, and then those are transformed and split into fragments, or pixels, that get the final rendering treatment before they are passed onto the screen.

-Deferred Rendering

Deferred Rendering is a technique used to optimize the rendering of scenes with a lot of light sources. Unlike traditional forward rendering, which computes lighting for each object in the scene individually, deferred rendering separates the rendering process into several distinct passes:

Geometry Pass: In this pass, the scene's geometry is rendered and various attributes of each fragment (such as position, normal, albedo colour, and specular intensity) are stored in multiple textures known as the G-buffer. These textures hold all the necessary data for lighting calculations.

Lighting Pass: In this pass, a screen-filling quad is rendered, and lighting calculations are performed for each pixel using the data stored in the G-buffer. This way, lighting computations are done once per pixel instead of per fragment per light source, which is much more efficient, especially in scenes with high-depth complexity and many light sources.

That is the generic implementation of Deferred Rendering, but it can also be used to perform a number of graphical techniques making use of its texture based pipeline. The information of the image is broken down into different textures made via passes (examples can be our SSAO technique explained below making use of a blurr pass alongside the already mentioned geometry and lighting passes).

-SSAO

SSAO or screen space ambient occlusion is a computer graphics technique that is used to enhance the realism in the lighting of a three-dimensional scene. It is used to simulate how light behaves in the real world, especially in areas where ambient light is partially blocked. This means that the points where two surfaces meet receive less ambient light, and therefore appear slightly darker.

This specific ambient occlusion focuses on optimizing the occlusion so that instead of being applied to the entire scene, it is only applied to what is visible from the camera.

To obtain this effect, first calculate the position and normal of the current fragment from the gPosition_ssao and gNormal_ssao textures. Then, a random vector of the noise texture is obtained, which is used to vary the directions of the samples. Using the random vector and the fragment normal, the TBN matrix is constructed, which transforms the samples from tangent space to view space.

Once this is done, the occlusion calculation has to be done for each sample. First, the sample position is calculated by transforming the samples from tangent space to view space and scaling them according to the radius. With this, the position of the sample is projected into the screen space. It is checked if the projected position is within the boundaries of the screen so that no artifacts are generated at the edges of the screen.

After iterating through all samples, the occlusion factor is normalized and adjusted to eliminate very low values and finally, the calculated occlusion factor is assigned to FragColor.

After calculating the SSAO texture, a slight blur is applied to remove impurities and sent to the scene paint shader to be applied to the ambient light of the final scene.
The rendering flow used was as follows: First in the SSAO calculation shader the SSAO texture is calculated and stored in a buffer. For this calculation we

send position, normal, noise, kernel, and screen size. Position is used to calculate the depth of objects on screen, normals are used to determine the orientation of surfaces and how they block ambient light.

Noise is applied to vary the samples' directions, which helps reduce artifacts and repetitive patterns in the calculation of ambient occlusion. The kernel contains several samples that are used to assess the proximity and orientation of nearby fragments. The screen size is needed to calculate the scale of the noise texture and to transform the positions of the samples to the screen space.

After applying the SSAO texture, the applied depth blur helps to improve the visual quality of the rendered scene, providing a smoother and more realistic appearance, and eliminating unwanted artifacts.

-Blur

The implemented blur effect performs a Gaussian blur in the depth space, with the aim of smoothing the image, considering the depth differences between pixels to preserve important edges.

First, a 3x3 Gaussian kernel is defined for the blur. The kernel values are chosen to sum to 1, which ensures that the image is neither lightened nor darkened. The current depth of the fragment is then obtained. Once the current depth is calculated, iterate through the Gaussian kernel, using a nested loop to cover the 3x3 texel area around the current fragment.

Here, for each position in the kernel, the corresponding displacement in the texture is calculated, and the depth of the sample, which is obtained from the displaced depth texture, is also calculated. The kernel weight corresponding to the position is calculated. Finally, if the depth difference between the sample and the current fragment is less than a defined threshold, the kernel weight weighted sample color is accumulated.

After accumulating the weighted colors, the result is normalized to ensure that the colors are neither darkened nor lightened and the defocused color is returned, to which the highlights are then applied.

-Portal Rendering

Portal Rendering works by rendering the scene from the point of view of the player if they to be placed at the other side of the portal. This is process that happens each frame to render a scene with portals:

-The current recursion level is set to the maximum of bounces allowed between portals
-An occlusion culling query is issued so the portals outside of the main scene view don’t perform useless draw calls
-A framebuffer is filled with the scene drawing taken from the point of view of the camera if it were to be placed at the other side of the portal, and cropped to the size of the portal
-The current recursion level goes down by 1
-From that scene render, the process is repeated so now every portal visible in the scene rendered in the framebuffer, also renders the scene
-This process runs indefinitely until either there are no portals left to render from in one of the views, or the current recursion level reaches 0 (if there were any portals visible from this last framebuffer, this are draw with a pink texture representing the missing texture).

-VR Rendering

The VR rendering pipeline, fundamentally, is based around drawing the scene two times from different slightly offset points of view, with different MVP matrices being used for each of the eyes rendering.

In our case, the following process is followed to render a VR scene:

Main VR view:
-A framebuffer is created for each of the eyes
-In each framebuffer, the scene is identically draw but each eye gets its own MVP (model, view, projection) matrix from the eye point of view.
-The controller Axis (rays that go from the controllers outwards) are rendered directly on top of each of the textures
-Similarly , the controller models and poses are obtained from .json binding files (located at *data/openvr*) which, automatically load all the information using the SDK backend
-The framebuffers are submitted to the VR headset as textures created by the VRcompositor, again, using the SDK backend

Companion Window view:
-The framebuffers are reused to render each of the eye views, displayed in a split screen manner. This setup replaces the typical rendering of the window.
-Then ImguiUI is then rendered on top of it

Strategies and Solutions to development problems

-SSAO

During the implementation of the SSAO, we encountered several problems. The main one was that when calculating the SSAO, patterns were created in the texture that kept repeating. This was because the kernel that was applied to get random directions was repeated over and over again. To solve this, another kernel was implemented which multiplies on top of the first one and a random start to the kernel. These changes meant that the pattern no longer repeated itself, as it became much more random.

-Blur

In the implementation of blur, we also encountered several problems. The main one was that due to the way we rendered the scene, we didn't have a screen texture with everything drawn, things were drawn directly on the screen. Because of this, we couldn't implement the blur independently with its shader. Instead, it had to be implemented in the paint shader of the deferred rendering.

-Shadow Mapping

An issue where specifically the shadow of the directional light has plagued the project since it was created. We had many shadow bias issues which were carefully maneuvered around, but the shadow of the directional light is entirely dependent on the distance to the camera both in forward and deferred rendering. To solve this issue the solution we found was making the perspective relation in the lightspace matrix sent being bigger to fit a small or medium scene, but we haven’t found a concrete solution for every case, specially when the camera is really far away from the shadow.

The correct solution would be the implementation of cascaded shadow maps, but that technique isn't implemented in our engine yet.

-Portals

Portal rendering is extremely expensive, so as mentioned in the combining Forward Rendering and Blinn-Phong part, we had to make changes to the way forward rendering is computed. We also tried to implement portals making use of deferred rendering, but that proved difficult as the pipeline to render portals was already created before we implemented deferred rendering, so we had to make a sacrifice and use forward rendering instead. As a result, some issues with forward rendering lighting have arisen and are still present in the final version of the engine.

Task distribution

Both:
-ECS
-Job system
-Camera
-Blinn-Phong Lighting
-Forward rendering

Carlos:
-Resource Manager
-Shadow-Mapping
-Deferred rendering
-Pseudoscripting subsystem & Playmode
-Portals
-VR implementation

Aimar:
-Billboards
-UI
-Scene Saving
-Input
-SSAO
-Blur