Robot Arm Report

A Computer Graphic Project

Denis Beqiraj, Lorenzo Ronzani, Elia Salmina
Computer Graphics Course - SUPSI DTI
Client: Peternier Achille

Fall Semester - Academic year 2021/2022
26. January 2022

Introduction

The project has been created for learning-purpose for the Computer graphics and Software Engineering courses. Every aspect seen in Computer Graphics classes must be in some way implemented in the project

The final goal was to create a 3D scene with a robotic-arm that moves through nodes around the scene with some features like catching a ball on the scene, moving it around, resizing it and dropping it.

This project in-fact helps us to apply the theoretical lesson and by practicing them we learnt how to make an efficient and consistent work by teaming and organizing the code structure.

Requirements

The requirements of the project was to create a generic engine written in C++ and using the OpenGL 1.1 graphic library, with an IDE of our choice.
Client and Engine must be separated, and the latter should be initialized only once at the start of the code then utilized in the client. The client must not see the logic behind each and every method, but can only utilize the methods and pass it the arguments.
The engine must be able to load any *.ovo* file and display it correctly, with all the textures, lights and meshes in the correct places.

To specify more, the scene must contain a segmented robotic arm with three sections that can be moved independently from each other. Two or more light sources and at least one movable and one static. Two or more cameras, following the same principles of the lights. How these elements behave and move it's up to us.
Additionally in the scene there must be some non-interactable objects and an interactable ball that can be caught with the robotic arm, compressed and released. Once released it must fall to the ground and stop at ground level.
Every object in the scene must have a planar shadow, as seen during the course, as well as some textures.
To create these objects we had the choice between picking third-party models or creating one by ourselves using 3D Studio Max. The latter also allows to create the entire scene
A FPS counter must be available at all times and it must always be higher than 24, together with a showable tutorial on all the buttons and shortcuts to navigate around the scene.

The final product must be able to run in both Windows and Linux as a standalone project, so a *.dll* or *.so* must be provided (depending on the platform). The engine creates the library and the client uses it.

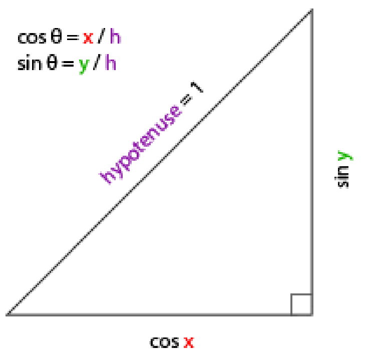
Additionally, everything must be developed with Design Patterns as seen during the Software Engineering course. And a SCRUM methodology for agile programming, using sprints to divide and finish each part of the project, must be used as well.

Architecture

The architecture is divided into client and server.

Client architecture:

The client architecture uses the dll and include files of the engine, it's needed to include only engine.h to get the whole interface, the first thing to do is to init the engine with handlers needed like the mouse handler or the keyboard handler, secondly we load the scene in node variable with Engine::Load, and as last thing we iterate until the user click a exit button, during the iteration we swap buffers, update, and pass the list of nodes into the engine, with the camera through method Engine::render, that's all needed to load a simple scene and show.
Our project is to create a robotic arm and move through the map so we bind the keyboard handler and go through nodes every time the user clicks “n” button, the “y” button simply make the application exit by a boolean variable, “p” changes the camera by putting the camera in 2 spots or letting it dynamic(as default) and if it’s static with “w” and “s” you can move over 2 cameras, if dynamic the "wasd" key make the user move through the scene by moving the cameraPos adding to it the cameraFront multiplied by a speed, for w subtracting for s,a and d keys moves right and left so how to move them?We apply a simple algebric rule that says that the cross gives us the perpendicular vector of the 2 multiplied vectors, we apply this rule on cameraFront and cameraUp so we get the perpendicular that needs to be normalized and multiplied by speed, we sum that to camera pos or subtract and we finally get a and d moved on left or right side.
The "ijkl" moves the current node forward,backward, rotates in clockwise direction or anticlockwise, “r” resets all variables and reloads the scene, “t” releases the ball putting a variable to true, and “c” or “v” takes cubes of fingers and bring them closes or farer, finally expands or restrict the ball, the “h” puts the help on or of drawing it in a position decided by user.
We have chosen the camera movement through the mouse because of the user’s comfort and effectively it’s more comfortable than the keyboard.
The dynamic camera starts from initializing the first position of that, secondly it sums an offset multiplied by our speed continuously and sums that offset to current yaw and pitch, and through Euler's angles it rotates on the scene.
Here’s a little explanation of euler angles:



If we define the hypotenuse to be of length 1 we know from trigonometry that the adjacent side's length is cos x/h=cos x/1=cos x and that the opposing side's length is sin y/h=sin y/1=sin y. This gives us some general formulas for retrieving the length in both the x and y sides on right triangles, depending on the given angle.
So in case we want to move through y axis we do:

direction.y = sin(glm::radians(pitch));

Similar for x and z, but we must take in consideration even the yaw, so we can rotate through that with:

direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));

and for z we calculate the sin instead of cos, we check if we don't get out of the screen by checking the values of yaw and pitch that mustn't be more than 89 degrees or less than -89 so we can move only by 180 degrees.**[1]**
After the handlers init and the loading of the scene we disable the plane shadow so we don't get a black plane because of it shadow.
In the loop we check if the ball is down, after that we check the distance of our center hand node and the center of the sphere taking the last column of the matrix so we get the coordinates and subtracting both center's and calculating the length.
If we let it fall it fell's until it touches the ground moving it down.
When we catch the ball we firstly add it as a son of the hand and remove it from the root node, similarly when it falls.
Finally we calculate the fps subtracting the start of the loop time and the end of it, we put it on screen every second.
All tests are made on the main function so when we run the application it tests that all works correctly.
The camera is created externally, so we have the camera object that is not in the scene itself and we calculate it with lookAt so we need to stay in alert because it returns the camera already inverted and when we calculate the scene we call inverseCamera so we calculate the inverse of the inverse and we get the original camera.

Engine architecture:

Object is the main class and every object that inherits the object class get a new id every time, node inherits over that and has all methods needed for the lights, meshes and material, with render method we render the scene recursively, the getFinalMatrix returns the internal matrix of the object and addChild, removeChild adds and removes childs from the node, we could set the parent with parent method or remove parent with removeParent.
Let's start from the init of the engine and methods, as first thing it's a singleton so it can be instantiated only one time, after these premise we init the basic glut function and set handlers, activate basic functions and we are using as basic ambient light 0.2 (because it's not provided by ovo to lights), clear clears the screen by black color, swap swaps buffers and update calls an update by glut, drawText draws text where the user needs to be drawn by providing x and y coordinates, we need to clear the screen before and disable lights, change projection type to ortho and when after drawing text we enable back all the functions and put the projection to perspective.

Going deeper we have the load function that returns the root node of a scene by an ovo file, it creates a objectloader, that as first inits the lights and textures, after that in load function it reads the ovo file, creates an initial position that will be used later to take index of the current position readed.
We dispatch object,materials, nodes, lights or materials using readNode function that returns an object called nodetype that contains the mesh,light,node(with the eventual number of sons) or material, all filled by reading with memcpy and creating objects, going deeper in that function:

- light fills his parameters with matrix, the current number of light and other parameters needed to the type of light after the subtype will be dispatched internally in light class

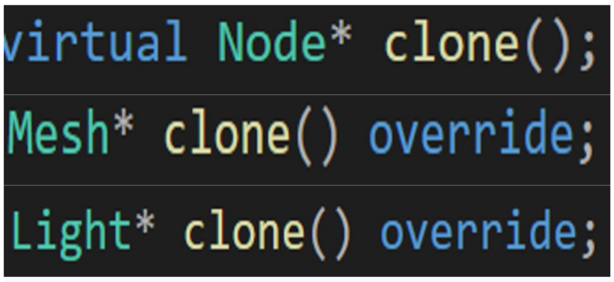
- node has only number of sons to fill and the matrix

- materials fill the parameters with the material id, texture name and multiply by constants to convert the ambient,diffuse and specular

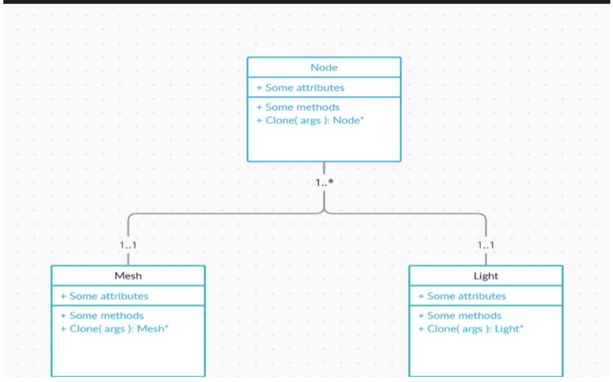
- meshes adds material if it has one and iterates through all lods and adds all of them in the LODdata struct that contains an array of VerticesData, inside that struct we have all information about vertices,normal,uv and faces.

Lights and texture ids are setted here by incrementing the value every time we get a new one.
The textures are loaded here and checks if the name of the texture is none, after that we bind the texture, set it in repeating mode and with mipmaps, we even use anisotropic filtering, after all we free the resources and bind the default opengl texture 0.
In it read materials, in that function we skip the first object file, and read all materials checking if materials are void or not, finally we put all these materials in a general map that we will use later to fill the meshes that uses these materials, after the iteration we get back by one because we read even the last position, so after the update of the position we read all lights and meshes recursively and add to the node, applying in the readNode the materials per mesh.
Finally we get our scene filled by nodes, and we need to render all of them so we need to pass the list and send it to the engine to render, the list is implemented by a dequeue so when we add a member to the list we dispatch if the type is a light or a mesh, in first case we put it before, else it will be putted after.
Our list has a method pass as the slides says that passes the whole scene and adds it to the dequeue that we iterate after, so what does this method do? It goes recursively through all nodes and adds recursively, multiplying the actual matrix to the son’s one, adding that to the dequeue, and passing the actual son that has the matrix updated, why is it convenient that method? Well because we know the type to dispatch directly and we don’t need to save externally.
The language blocks a necessary operation, the deep copy of the pointer, because we have a pointer and C++ hasn’t virtual constructors when we deep copy the node it doesn’t copy the references to mesh or light so we have adopted a pattern that’s called virtual copy pattern or virtual constructor, what it does? It creates a method called copy that needs to be inherited and returns a copy of the object itself, why does it work? The override returns the

pointer to the complete type of the object so when we copy the object obtained returns the effective deep copy.



```
virtual Node* clone();
Mesh* clone() override;
Light* clone() override;
```



Finally we get the List and we can pass it to the Engine::render that needs the list and the camera.
The engine iterates over all the list and calls render.
Light render:
The light render checks if we don't exceed the max number of lights, after that checks the type of light and sets the opportune settings, the binding of the light is dynamically chosen by the sum of the basic id given by objectloader and the offset given by light 0, diffuse and specular are decided by color setted in 3ds max, the light is moved through his current matrix, so we load it in opengl by multiplying it with camera
Mesh render:
As first thing sets the lod to 0, multiply the camera inverse with final matrix and bind that, so we check if the mesh has material or texture and binds it, else it set to default opengl texture 0 and the elements that hasn't a texture will get the default one, after that we copy the vertices data so we don't need to access to it every time, it improves by 30 fps the performances, after that we iterate through vertices, we copy the 3 vertices position in faces and get 10 more fps, after that we get the position by faces so every face has the positions of the corrispettive vertices,normals,uvs and we set the pointer through opengl functions.
As last thing if the mesh has shadow we render it by multiplying the identity matrix with y=0 with final matrix and after we do camera matrix inverse multiplied with the shadow new matrix, we disable textures, lights, do other improvements copying references so we get other 15 fps and render the scene, in the end we enable textures and lights.
The at function obviously is slower than [] overload of std::vector, but it grants a layer over the bound checking and various tests in our code have checked that it doesn't slow the execution.
Some utilities that we have added are the recursive search of a node that returns by a name in input the reference to the node.
The search by a map on the List so we can get the references.

The enable and disable shadow so we can avoid problems with the plane that it shows completely black because it overlaps the shadow.
The camera could be external so we can use a camera created by the program or we can take an object over the scene and move the final matrix.

Testing

During the development we have tested mainly the engine component, because it is the main core of our application.
We created a test class that tests all fundamental parts of the engine class.
Firstly we test the correct binding of the list class, we add to the list object 2 types of element (Light and Mesh), and we check that lights are on the top of the list, this is so important for the render phase.
Secondly we load a simple scene, we take the plane (that is a child of the scene tree), and we check that the material is loaded correctly.
Thirdly we take the number of children of the root node, we remove 1 child and we take again the number of children, this to check the functionality of removeChild and getNumberOfChildren.
In the same way we also test the operation of adding a child.
Finally we test the getFinalMatrix operation, we take the matrix of a sphere in the list object and we compare it for the real sphere matrix.
Our engine passed all tests for these critical operations.
The tests were essential to check the correct functioning of the engine.

Results

We have developed a fully functional product, tested with the test class previously specified.
We have developed a generic graphic engine that is able to load a scene from an .ovo file and render it in an opengl window.
Our engine works both on Linux and Window, this is possible using the precautions provided by prof.
We used our generic engine to create and control a three segment robotic arm that is able to interact with one ball, catch and release them.
The ball has a fake gravity that allows us to reposition the ball on the ground.
Our game has 4 lights:

- omnidirectional, that illuminates all scene, it is like an ambient light
- fixed spot, that illuminates the base of robotic arm
- mobile spot, that rotates automatically throughout the scene
- mobile spot, positioned in the top of robotic arm, that illuminates the direction of robotic arm

Our game has 3 cameras:

- mobile cameras, it can move with mouse
- 2 fixed cameras, positioned on the walls, that allow us to see the scene to another perspective.

Our engine is able to render shadows of objects, with the fake shadow technique, previously explained.

We used the technique “code a little test a little” to develop a functional product, as well as utilizing the SCRUM methodology that allowed us to better separate each and every task for every member of the team

References

https://learnopengl.com/Getting-started/Camera [1]