POLITÉCNICO DE LEIRIA

Computer Graphics

2024/2025

Project II

Xenon Render

By:

2221942 | Damien Ribeiro

2231864 | Afonso Alexandre

November 29th, 2024

Contents

Introduction

In this first project of the Computer Graphics curricular unit, we were tasked with recreating, using OpenGL, the rendering of the Xenon 2000 game.

Implemented

Illumination Models

The Phong and Gouraud models make use of three components in order to calculate the color of a surface.

Ambient Lighting, simulating indirect lighting from the environment. This component can be calculated by multiplying the Ambient reflectance (a property of materials) and the color of the ambient light.

Diffuse Lighting, simulates the scattering of direct light on a light scattering surface. The closer to normal the light hits the surface the brighter the surface will be. It can be calculated by multiplying the Diffuse reflectance (property of material) with the color of the light and multiplying with the dot product (how close two vectors align direction wise) of the surface's normal vector and the light direction (both normalized), this product must be clamp to zero as the surface is at its darkest when the light comes from behind it .

Specular lighting, simulates the highlights from direct light reflection. It can be calculated by multiplying the specular reflectance (property of material) with the color of the light and multiplying with the dot product of the reflected light vector, and the view direction, clamped to 0 . This dot product is then raised by a shininess coefficient, resulting in sharper highlights the higher the value is.

The final color of the object can then be calculated by adding them all together.

*Ambient lighting = Ambient light color * Ambient Coefficient*
*Diffuse lighting = Diffuse light color * Diffuse Coefficient * clamp (0, Surface Normal.Light Direction)*

*Specular lighting = Specular light color * Specular Coefficient * clamp (0, Reflection Direction.View Direction)^shininess*

Color = Ambient + Diffuse + Specular

Considering texture, we have to multiply it with the overall color.

*FinalColor = Color * Texture.*

Differences between Phong and Gourand.

Gouraud has its calculations performed in the vertex shader, while Phong has them run in the fragment shader. This results in Gouraud being less performance heavy, as it only calculates lighting in the vertices, but can result in inaccurate specular highlights and less realistic lighting effects due to its per polygon interpolation of the color values.

Phong provides more precise lighting for each pixel (fragment) resulting in smoother and more accurate results. It interpolates surface normals across the polygon in order to deliver precise lighting.

Both these models are implemented in this project.

Wavefront Loader

Once called, the wavefront loader reads the .obj file twice: first it keeps count of how many vertices, texture coordinates, normals and triangles are in the file – it does this by reading each line and checking if the first word of the line is either "v" (for the vertex positions), "vt" (for the texture coordinates), "vn" (for the normals) and "f" (for the triangles). Afterwards, if one of those values was found in the current line being read, it adds one (1) to its respective count. Once every line in the file is read, it then reserves the same value of each counter on each of the vertex data vectors.

On the second time the file is read (it reads it in the same way and also compares the first word of each line), it populates each vector with its corresponding data taken from its respective line.

While the way the vertex, texture coordinates and normal vectors get populated is very linear, the "vertices" vector (the one that stores all the vertex data (vertex position, texture coordinates and normals) in the order they should be rendered) first reads each of the model's triangles' vertex data (or "corner"). Meaning, it first stores the vertex positions, then texture coordinates and finally the normals.

Vertex Buffer Object Indexer

The VBO Indexer works by taking a vector of vertices (with vertex positions, texture coordinates and normals) already in order (one could use *glDrawArrays* to draw the vector's model perfectly) and separates it into two vectors: one containing only the unique vertices present in the input vertices vector, and another storing the order of which those unique vertices should be drawn.

To achieve this, it first "packs" the input vertices by taking, at a time, eight values stored in that vector and copying them into their own vertex vector which then gets added into a vector for storing all the packed vertices. Then, once all the vertices have been packed, each packed vertex gets compared to a vertex already stored into the unique vertices vector

and, if there isn't a similar vertex in that vector yet, it "unpacks" the vertex back into separate values, adds them to the unique vertices vector and stores a new index value. If it does find a similar vertex however, it gets the index of that vector which was already added, and only stores that index instead.

Model Loading

For each model the user tries to load, the user must initialize a new *Object* by specifying the file paths for the wavefront and texture files and vector of objects to be drawn and its count. Then, inside the *Object* class, the wavefront loader is first called, then the VBO indexer gets called to index the vertices outputted by the wavefront loader, and finally the objects to be drawn counter goes up and reserves more space in the objects vector accordingly. The only extra step the user has to do afterwards is adding that new *Object* into the objects' vector.

After the user loads all the models, for each model in the objects' vector, a vertex array object and texture are created and the VBO and IBO sizes of the object get added to the shared VBO and IBO sizes. After the shared VBO and IBO are created with all the objects VBO and IBO sizes added up, a buffer subdata (containing each object's VBO/IBO size and offset in the respective buffer) is added to each buffer. Next, each object's VAOs are finally generated and bound one by one to set all the vertex attribute pointers according to each of their offsets.

Finally, inside the render loop, each model's corresponding VAO and texture are bound and the model's position, rotation and scale is set before the model is drawn.

Camera

The camera is implemented similarly to the way we did in class, except that translation of the camera now instead of using the vector of the direction the camera is looking at (*front*), it uses a projection of the direction vector on the horizontal plane (x, 0 , z), that we call horizontal front (*hfront*).

Objectives

Achieved

Model Loading

Shader Programs

Matrix Transformations

User Interactions

Unachieved

All objectives were achieved.

Decisions

Shader Programs

Originally we started making the shader programs very similarly in structure to the ones made in class. During our search of possible basic illumination types we could implement we found various examples that had their shader files ready for input of material properties. Due to this we ended up adapting the Phong shader and making the Gouraud shader in a similar fashion.

User Interaction

By reading the assignment, this task seemed to be mostly covered by the implementations we made in class. We decided to expand on it, handing the user control not only over the camera (implemented like in class), but over an object representing a light, so the user can see how the shaders react to the moving light. But also control over what shader is being used (Phong or Gourand) and the material properties of the objects in the scene.

The full list of inputs can be found at the end of this document.

References

- Model Loader
- VBO Indexer
- Drawing Multiple Models
- Basic Lightning
- Gouraud Shading
- Diffuse Light (Jamie King series)
- Gourand and Phong example

Attachments

Control Scheme

Category	Control/Option	Option	Key
Camera Controls	Movement		
		Move Forward	W
		Move Backward	S
		Move Left	A
		Move Right	D
		Move Upwards	E
		Move Downward	Q
	Zoom		
		Zoom In	Mouse Scroll Up
		Zoom Out	Mouse Scroll Down
Light Source Controls	Positioning (relative to world origin not view)		
		Move Forward	I
		Move Backward	K
		Move Left	J
		Move Right	L
		Move Upwards	O
		Move Downward	U
Lighting	Lighting Keys		
		Ambient Lighting	NumPad_1
		Diffuse Lighting	NumPad_2
		Specular Lighting	NumPad_3
	Color Keys		
		Red	R
		Green	G
		Blue	B
	Color Values		
		Increase	Lighting Key + Color Key + UP Arrow

Category	Control/Option	Option	Key
		Decrease	Lighting Key + Color Key + DOWN Arrow
	Coefficient Values		
		Increase	Lighting Key + Color Key + RIGHT Arrow
		Decrease	Lighting Key + Color Key + LEFT Arrow
	Other Lighting Keys		
		Increase Shininess	NumPad_0 + UP Arrow
		Decrease Shininess	NumPad_0 + DOWN Arrow