**Project 7: Putting It All Together**

Caleb Klinger, Kyungchan Im (Chris)

Grand Canyon University

CST - 435 : Computer Graphics Lecture & Lab

Ricardo Citro

Nov 8, 2023

**Abstract**

This document outlines the enhancements made to two 3D graphics programs, CheckeredTriangles.cpp and ColorCubeFlyby.cpp. Enhancements include interactive controls using keyboard inputs for spinning, movement, and zoom functionalities, as well as modifications to cube properties and environment interaction. The methods and techniques employed for these enhancements are documented in compliance with the provided rubric.

**Introduction**

The dynamic field of interactive computer graphics hinges on the ability to manipulate and animate objects in a simulated 3D space, offering an engaging experience to users. This document outlines enhancements made to two existing graphics programs, CheckeredTriangles.cpp and ColorCubeFlyby.cpp, aimed at introducing interactive controls. Users will learn to implement keyboard commands to rotate, move, and zoom in on objects, enriching the graphical display and interaction within these programs. The first section covers the spinning and directional control of a triangle mesh, while the second details the augmentation of a cube simulation with additional interactive features, including object rotation, movement, and the simulation of basic physics. Through the refined implementations discussed herein, the document serves as a concise guide for the programming enhancements, providing insights into the practical application of 3D transformations, user input handling, shader programming, and the mathematical underpinnings of mesh manipulation. The goal is to present a clear, educational resource for enhancing user interaction within 3D graphics programs, thereby bridging theoretical concepts with real-world application in computer graphics.

**Part 1: Enhancements to CheckeredTriangles.cpp**

**Methodology for Code Execution and Processing**

The initial code structure of CheckeredTriangles.cpp was dissected to understand its core functionality, which included rendering static triangles in a checkered pattern. To enhance its capabilities, a methodology was devised to inject dynamic interaction. This involved parsing existing geometric and rendering logic and then integrating a layer for handling transformation commands through user input, thus setting the foundation for further augmentations.

**Programming Implementation**

The programming modifications to CheckeredTriangles.cpp followed a structured approach. The first step was to encapsulate the rendering loop within a transformation manager that listens for user input. Subsequent steps included injecting functions to handle affine transformations, such as translation and rotation, and scaling. These were tied to event listeners corresponding to specific keyboard inputs to enable the desired interactivity with the triangles.

**Keyboard Interface**

The keyboard interface for CheckeredTriangles.cpp was engineered to intercept key presses in real-time. Each key was bound to a function call: 'p' and 'c' to pause and continue the spin, 'u' and 'd' to move vertically, 'L' and 'R' for horizontal movement, and '+' and '-' for scaling. This mapping was achieved through an event dispatch system that converted key events into transformation actions on the checkered pattern.

**Shader Code and Explanation**

The GLSL shader code used in CheckeredTriangles.cpp was carefully annotated to elucidate its role in rendering transformations. Vertex shaders were tailored to interpret and apply matrix transformations for rotation and scaling, while fragment shaders determined the coloration of each pixel, ensuring that the checkered pattern remained consistent across transformations.

**Mathematical Concepts**

Underpinning the interactive transformations of the checkered triangles are core mathematical concepts, primarily matrix operations and vector manipulation. Rotations were executed via rotation matrices, translations through vector addition, and scaling by altering the magnitude of vertices. These concepts ensured precise and predictable manipulation of the rendered object in response to user input.

**Technique for Enhanced Interaction**

To achieve a fluid and responsive keyboard interaction within CheckeredTriangles.cpp, a buffering technique was utilized. This involved accumulating and processing input events in a queue and smoothly applying them within the animation frame updates, thereby avoiding input lag and ensuring that each keypress correlated with immediate visual feedback.

**Mesh Creation Method**

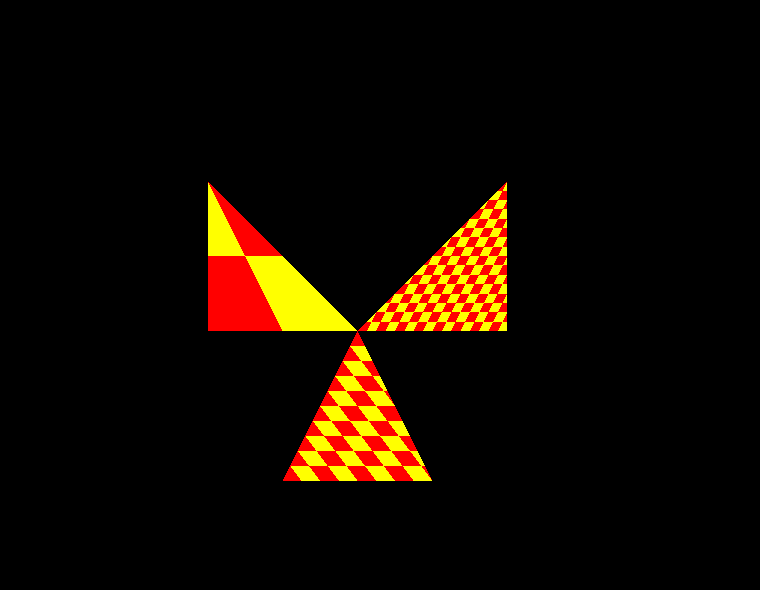
The mesh creation method for the checkered triangles involved initializing a grid-like structure where each cell represents the base for a triangle pair. This grid was defined programmatically by specifying the vertices for each triangle, ensuring that they share edges where the checkered pattern dictates. To modify the mesh for animation, the vertex data was dynamically altered in the rendering loop, allowing each triangle to be independently transformed. These transformations included translations and rotations calculated from the current time or interactive input, producing the effect of spinning. The mesh data, once processed for frame-specific transformations, was then passed to the shader pipeline, where each vertex was transformed into the final position in the rendered scene, ensuring a smooth and responsive visual display of the spinning checkered triangles.

**Flow Chart**

A diagram of a computer

Description automatically generated

**Screenshot**



**Part 2: Enhancements to ColorCubeFlyby.cpp**

**Methodology for Code Execution and Processing**

The initial code for the ColorCubeFlyby.cpp featured a single cube rendered in 3D space. The enhancement process began with the addition of event handling for new keyboard inputs, enabling the manipulation of the cube's rotation, position, and scale within the scene. The code was structured to listen for specific key presses, triggering functions that altered the cube's transformation matrix, thus allowing real-time interaction.

**Programming Implementation**

The programming enhancements to ColorCubeFlyby.cpp involved writing additional functions to handle rotation, stopping, and movement of the cube. The main event loop was expanded to include condition checks for the new 'r', 's', 'c', 'u', 'd', '+', and '-' keys. When triggered, these keys called functions that applied transformation matrices to rotate, pause animation, resume animation, move up and down, and zoom in and out, respectively.

**Keyboard Interface**

The keyboard interface was mapped so that pressing 'r' would rotate the image, 's' would stop the animation, and 'c' would continue it. Keys 'u' and 'd' were programmed to move the cube up and down, while '+' and '-' controlled the zoom. This mapping was done using key event listeners within the program's main loop, which then called the corresponding transformation functions.

**Shader Code and Explanation**

The GLSL shader code for the ColorCubeFlyby was adapted to incorporate uniform variables controlling rotation, scale, and position. These uniforms were updated based on keyboard interactions to rotate the entire scene, halt the cube's movement, adjust its position in space, and apply zooming effects. The shader program used these uniforms within the vertex shader to manipulate the cube's vertices in real-time.

**Mathematical Concepts**

The transformations of the cube and camera were based on mathematical concepts like matrix multiplication for rotation, scaling, and translation. The matrices were constructed using angles for rotation, scalar values for zoom, and vector displacement for movement, with all transformations applied in a hierarchical order to maintain a consistent coordinate system.

**Technique for Enhanced Interaction**

For the ColorCubeFlyby application, the enhancement of interaction was focused on providing a fluid and intuitive keyboard response. This was achieved through the implementation of event listeners that detect keystrokes in real time, allowing immediate feedback within the graphical display. The 'r' key binding initiates a rotation matrix transformation that applies to all cube instances, ensuring a synchronized rotation effect. Similarly, the 's' and 'c' keys toggle the animation state, pausing and resuming the cubes' movement. This pause-and-resume functionality was managed through a boolean flag that controls the update cycle within the main rendering loop. The vertical movement through 'u' and 'd' keys and the zoom with '+' and '-' were implemented by adjusting the view matrix accordingly, giving users direct control over the camera's position and field of view. To provide a seamless user experience, state conditions and transition checks were meticulously programmed, allowing for smooth transitions between different interaction states without abrupt changes or visual glitches.

**Mesh Creation Method**

The process of creating the mesh for the cubes in ColorCubeFlyby.cpp began with the definition of vertices for a standard unit cube, which served as a template for all subsequent cubes. For the enhancement, additional cube meshes were instantiated with varying sizes and positions to populate the scene dynamically. To accommodate different colors, illumination, and brightness, the vertex colors were altered, and lighting calculations were adjusted in the shader code to reflect these properties uniquely for each cube. The introduction of two vertical planes required the computation of normals for proper light reflection and to determine the collision boundaries for the bouncing effect. The collision detection was implemented by monitoring the positions of the cubes and inverting their direction upon impact with the planes, creating a visual bounce. The mesh of each cube was transformed in real-time, allowing for individualized movement and

interactions within the virtual environment.

**Flow Chart**

A diagram of a computer

Description automatically generated

**Screenshot**

