CS 174A Final Project - Cube Field

Introduction

Our project is a modern recreation of the classic arcade game Cube Field. The goal of the game is to navigate through the field of cubes without a collision and survive as long as possible. The player moves left with the semi-colon key and right with the apostrophe key. The User's current score and high score are displayed in the bottom left of the screen.

Player Object

The Player is generated using a for loop and the insert_transformed_copy_into() function. Within each iteration of the loop, a cube that is scaled by one tenth in the z direction is stacked onto the previously added scaled cube. This results in the pyramid shape resembling an arrow that represents the player. The player is given a random color every new round and has the default maximum ambient lighting to make it easier for the user to see the player.

Obstacle Object

Each obstacle is a simple unit cube. The cubes are randomly generated with a positive z axis value of 60. As time passes, the cubes' z value decrements to give the user the impression that they are moving forward through the field of cubes. Each obstacle has zero ambient lighting, minimal diffusivity, and maximum specularity. This gives the user the impression that there is very low lighting in the game field and creates an intense reflection off the near side faces of the cubes. The overall effect is similar to a car's headlight where you can primarily see only what's in front of the player object. This will be discussed further in the Camera and Lights section.

The obstacle generation is managed within the manage_obstacles() and add_obstacle() methods. In the manage_obstacles() function, an obstacle is added if enough time has elapsed since the last obstacle was added. This interval was determined through trial and error to make the game most playable. As the user survives longer, the interval is decreased which results in objects being created more frequently to increase difficulty. The initial value on the x axis (left to right with respect to the player) is generated randomly between -100 and 100 of the player's current x value. This results in an even creation of obstacles that always generate with enough spread to prevent users from moving quickly to the side to essentially outrun the generation of obstacles altogether. Once an obstacle's z value indicates it has travelled off the screen behind the player, it is deleted from the array which holds all the obstacle transforms. The add_obstacle() method simply pushes a new transform onto the array.

The funnel shape at the beginning of each round is simply a collection of obstacles generated with two for loops. Each loops creates one side of the funnel. The funnel behaves just like the other obstacles whereas the player cannot make contact with the wall without dying. The purpose of the funnel is to signal the start of the round and allow the user to get a feel for the player's movement characteristics before entering the obstacle field.

Camera and Lights
The camera transform and single light source move in unison with the player along the x axis to create the headlight effect described above. This effect was added to make the game more interesting and difficult as moving very quickly to the left and right results in a greater potential of running into an obstacle that is very difficult to see. The camera always looks straight down the positive z axis with a small y axis position to give the user a slightly raised perspective over the obstacle field.

Explosion Object
The explosion object is a Subdivision_Sphere object, with 3 subdivisions; this object approximates a sphere by starting from a tetrahedron and adding new faces by adding edges between midpoints of the edges of the original faces. With only 3 such iterations, the object does not look very spherical when flat-shaded. As such, we use Phong shading to render the explosion without needing many polygons.

The explosion object is essentially a sphere that rapidly grows until the maximum "blast radius" has been reached. In addition, the sphere is initially colored white, and uses a slightly modified Phong shader that "redshifts" the color over time. The pixel's color, starting out at (255, 255, 255), has its blue and green components decrease over time (and at certain rates relative to each other, in order to have the explosion's color cycle from white, to yellow, then to orange, and then to a dark red.).

In summary, the explosion is an initially small and fully white sphere, that grows over time as its colors cycle from bright white to dark red.

Physics - Player Movement
At the beginning of each frame, the input state is stored as a two-bit variable, *dirs.* Here, the left bit corresponds to whether or not the left key is pressed; the right bit, likewise, corresponds to the state of the right key. Note that *dirs* will be affected by both keypress and key-release events, with either bit being unset when the corresponding key is released.

After retrieving the input state, the horizontal velocity of the player is adjusted accordingly. The idea is simple: use this frame's input state to determine the player's "target" velocity. If either one (but not both) of the keys are pressed, then the target velocity is simply the maximum velocity (defined as a constant) in the corresponding direction. If both keys were pressed, or if neither key was pressed this frame, then the target velocity is 0. We obtain the result concisely with the following expression:

((dirs == 1) – (dirs == 2)) * max_velocity

Once the target velocity is obtained, we then "nudge" our old velocity towards the target by a small, constant amount (defined as acceleration in our program), clipping if necessary, to obtain this frame's current velocity. Once that is obtained, we then calculate the player's new position by adding the current velocity to the player’s x-coordinate. However, before the player can be moved to that position, we must check for collisions.

Collision Detection

We use a simple axis-aligned bounding volume method to test for collisions. That is, we give the player a cuboidal “hitbox” --slightly narrower than the actual player model to give the user an advantage--and use it to check for axial overlaps between the player and obstacles. The idea is this: if two cuboids are overlapping, then the overlapping region must form a third cuboid, whose dimensions are the overlaps along each axis respectively. We calculate whether these dimensions are non-negative, and hence, determine whether the objects are actually overlapping.

Once a collision is detected, the game’s state is updated accordingly, so that the explosion is rendered and the player dies, prompting the user for a restart.

Member Contributions

August - Initial setup of the game and files
- Player, Obstacle, Funnel, Camera, and Light objects
- Very basic Player, Camera, Light, and Obstacle movement
- Random Obstacle creation
- Presentation Video

Ambareesh - Physics related to Player movement
- Addition of Start screen vs. Actively playing screen
- Improved obstacle management including deleting off-screen obstacles
- Status Messages on Control Panel
- Collision Detection
- Explosion Object and Custom Shader

Both members contributed equally to the report.