Tron LightCycle game in WebGL

Marco Favorito 1609890

Table of Contents

Introduction

The project consists in an implementation of the Tron LightCycle game. It is inspired from a scene of the film *Tron (1982)*, where the main characters and the enemies have to compete in a game with their special motorbikes.

The game can be explained as follows: Two or more players are represented by dual-wheeled motor vehicles, or light cycle, in a grid-lined arena. The vehicles constantly move forward, leaving a coloured trail behind them as they travel. Contact with either the arena walls or a trail left by a player will result in death and elimination from the battle. Therefore, players attempt to box each other in order to force their opponents to touch their trail or one of the arena walls. Players can change the direction of movement, by turning 90 degrees to the left or right.

Environment, Libraries and Models

The **environment** is pure WebGL.

The used **libraries** not developed by the team are:

- *cuon-matrix.js* and *cuon-utils.js* from the WebGL Programming Guide (for linear algebra operations);

- a modified version of the *OBJReader.js (*for load the model defined in a *.obj* format) taken from an example of the WebGL Programming Guide;
- the usual *webgl-utils.js* by Google and *webgl-debug.js* from Chromium (for the WebGL APIs).

The only external **model** not defined by the team is the wheel of the light cycle (*/objects/future-wheel/wheel.obj*) downloaded from https://free3d.com/3d-model/future-wheel-10-93869.html.

Also the texture for the grid walls has been downloaded from internet.

Technical aspects

Lighting and Texturing

The only lighting in the scene is provided by an ambient light and a point light, which illuminates the scene from a certain height, on the center of the grid. The computation is made by fragment. The reason why it is not used a directional light is that the squared arena has four walls along its sides, which means that any direction will darken some wall.

The shader use both lighting and texturing with the following approach: if an object has to be drawn with only specified colors and no textures, then a “white pixel” texture is loaded and multiplied to the color (hence the texture does not contribute to the color); otherwise the appropriate texture is loaded and with the associated vertices white colors are loaded in the attribute buffer.

The texture of the grid, generated programmatically, a black square with white margin, so that when it is repeated, in order to cover all the ground of the scene, the arena become grid-lined.

The texture for the grid walls is made with an image of electric circuits repeated in the same fashion of the ground.

Other texture are used as “colors”: they are constituted by a single pixel. It is needed in order to draw the same shape with different colors.

Light Cycle Model

The light cycle model is constituted by the following parts:

- The main body (a parallelepiped);
- two wheels (a sort of cylinder), right and left. They rotate at the same speed during the game;
- the tail, which “generates” the wall;
- the head (a cube);
- a sort of monocle that is attached to the head and moves accordingly to it.

The model is drawn as a hierarchical model, starting from the body and "pushing-popping" the model matrix, as usual. The animated parts are the wheels (which rotate at constant rate) and the head with the monocle, which is animated when the player changes the camera direction (more details later);

Buffer organization

The system uses two sets of buffers:

- the "static" one, containing all the data which do not change during the game. It includes the vertices for the grid and walls (a square opportunely transformed), for the parts of the light cycle (a cube) and the wheel object. It is created with the flag STATIC_DRAW and the elements are drawn through indices (i.e. using "drawElements()").
- The "dynamic" one, containing data which changes often. It is the case of all the left light wall created by the light cycles. The same approach is used for the explosion animation, since it is used only for a small number of times. It is created with the flag DYNAMIC_DRAW and the elements are drawn as arrays (i.e. using "drawArrays()").

The main reason for this approach is due to the following considerations:

- Too many "draw()" call make the system very slow. It is the case when the left walls needs to be draw using only one shape (i.e. a scaled, rotated and translated square). Hence, we need to load vertices for every passive light wall, but...
- … Since the creation of new walls is quite frequent, load too many times the same data with only small updates is inefficient since it invalidates the cache of all the present data. For many shapes (e.g. cube, wheel etc.) this frequent load operation may lead to bad performances due to the relatively high number of vertices.

With the approach described above, this problem should be solved by just binding the appropriate set of buffers whenever they are needed.

Active and passive light wall

The *active light wall* is drawn by scaling a square at every step of the light cycle, so it seems to be generated from the starting point of the trace to the current position of the motorbike.

When the light cycle turns left or right, a *passive light wall* must be created. The active wall must be saved into the "dynamic buffers" described above; hence we pick the starting point and the position when the light cycle turned and add this data (with the associated normals, colors of the light cycle and "white" textures) to the dynamic buffers, which are updated. These buffers are used until some other passive wall need to be drawn from a certain moment until the end of the game or until the player who generated the wall looses.

When a player is eliminated, its active and passive walls need to be deleted as well. In order to remove the passive light wall of a certain player, we need to resize the buffers removing all the data of their walls. To make it possible, when adding new walls a dictionary “cycle to indices” is maintained, so that it is easy to remove all the data belonging to a player.

Collision checking, acceleration and AI

This three tasks (i.e. collision checking, acceleration and AI decisions) depend on the same approach.

The collision check for every cycle is done at every animation step. The horizontal and vertical axis (looking from the top) are computed accordingly to the model matrix, and for the segment relative to each wall the intersection is checked. If yes, then the light cycle is eliminated.

More precisely, the checks made are:

- vertical/horizontal axis vs passive walls
- vertical/horizontal axis vs active walls
- vertical/horizontal axis vs grid walls

In order to check if a light cycle needs to be accelerated (i.e. check whether a light cycle is running next to a light wall), I used a similar approach as before: I defined a new segment over the light cycle model like the horizontal axis but this time “expanded” on both sides, such that using the same algorithm it is easy to check if this “virtual axis” is intersecting with some wall. If this check is successful, then the light cycle speed is increased up to a maximum; otherwise the speed is decreased down to a minimum.

The AI engine uses the same approach, but this time translating the axis ahead in the light cycle direction in order to predict the future state. If there is any collision then turn.

Camera

During the game, the default camera looks at the player’s light cycle from behind. The offset between the camera position and the light cycle model is not set statically. At every time there is a “target position” and a “current position” for the camera: the distance between the first and the model is constant. At each animation step the discrepancy between this two positions is computed and the current position is modified accordingly to the difference but scaled with a small coefficient (e.g. 0.08), in order to smooth the animation and kindly make the transition.

When the light cycle turns, only the target position is rotated accordingly to the move. The “chase” of the current camera position to the target one is made in the same way.

Overall there are four modes that the player can use:

- CHASE: the camera “chase” the light cycle from behind. This is the default mode.

- LEFT/RIGHT/BACK: the camera points in the chosen direction w.r.t. the light cycle, it is located on the opposite side of the light cycle and looks at the light cycle. It is more elevated w.r.t. the CHASE mode.

The schema target-current position remain the same.

Especially for the last three modes, if the camera position is outside the grid some changes are made (i.e. the position is limited by the wall grid and look-at vector does not point on the light cycle but it is shifted towards the opposite side).

When the camera mode is changed, the head of the light cycle is animated. The animation consists of rotating the head to a target position according to the camera mode.

At every turn, the camera mode is set to “CHASE”.

Explosion

When a light cycle crashes, an “explosion” animation is started. The light cycle model is removed and another buffer is created. The new buffer contains a vertex for the center of the explosion and a dozen of points arranged in a circular shape, with a small height from the ground, which represent the fragments of the light cycle. The animation is implemented by drawing lines between the center and the other vertices and by scaling the model matrix: at the beginning until a certain point the vectors are enlarged in every direction, simulating the spread of the fragments. Then the only expanded direction is the y-axis, such that the rays converge in a single vertical line. Then the animation ends and the buffers are deleted. The color used for the lines is the same of the eliminated light cycle.

User Interactions

Start menu

At the beginning of the game, the user sees a simple menu in which he can read how to play and set some configurations (i.e. the size of the arena from 200 to 1000 and the number of enemies from 1 to 3). Then he can hit the “Play button”. The loading of the models and the buffer is triggered.

Start animation

At the beginning of every match the arena with all light cycles is shown. Each light cycle is located at different sides of the arena. A countdown starts and it is shown on the center top of the screen (3… 2… 1… Go!). The camera looks at the player’s light cycle from a higher position, far to the light cycle. During the countdown the camera goes towards the light cycle and is positioned to its back, waiting for the countdown end.

Gameplay

The light cycles start to move forward and never stops. The user can control his light cycle and the camera through the keyboard. The available commands are:

- A/D: turn left/right;
- ← → ↑ ↓: set the camera mode to LEFT/RIGHT/CHASE/BACK;
- P: pause the game;
- ESC: return to the start menu.

When the player light cycle crashes, the message “Game over!” is shown on the top of the screen and after some second the match is restarted.

When all the enemies are eliminated, the message “You win!” is shown on the top of the screen and after some second the match is restarted.

Further improvements

- Use of HUD for show the names of the vehicles during the game;
- Improve the AI engine so that a single light cycle finds a good plan, or even coordinated with the other light cycles;
- Introduce sounds (distance sensitive);
- Optimize the check for collision with hashing (i.e. hash the walls according to their coordinates and checks only the “candidate walls”).