CatRun

Jacopo Fabi 1809860

09/25/2022

The project is a endless running video games devoleped with Three.js , a JavScript library, and Tween.js , a JavaScript engine for animation. This is the final project of the course Interactive Graphics, Sapienza 2021-2022.

1 Gameplay

The way is builded with two different blocks "Room" and "Transition Room". "Rooms" are the blocks that form the linear path while "Transition Room" are used for change direction. "Rooms" can randomly spawn obstacles which fraquency increases as the game time increases.The game end when the animal hits an obstacle or a wall.

2 Game building

At the beginning, all the objects that will be needed during the game are created: obstacles, rooms, decorations and animals. These are immediately added to the scene and will never be removed from the scene. So we have a constant occupied memory space, therefore not very optimized, but we will not have variations in fluidity deriving from removing objects from the scene, or destroying and recreating them. When not needed, objects are made invisible and when they are reserved for the game, they are made visible.
To build the path blocks are joined one to another and the maximum number of blocks in the game is 6 at time. The animal in game has a constant z while blocks move in this direction with a speed different through the game.
When a block reaches a position of the zetas no longer visible from the camera it is eliminated from the game (i.e. it is made invisible) and another block is added to the farthest block.
Obstacles are created directly within the room when it is also created. By default they are invisible. When a room is added to the path, the type of obstacle that will be visible is randomly chosen.
The decoration lights aren't made invisible. When a light is removed from a scene or it is made invisible Three.js may compile shader and this is an expensive operation. To avoid this the decoration lights are always visible and on the

scene but positioned, if they should not be in the path, in an area not visible from the camera. When a room need a light(random choice), the position of the light will be matched to that of the room and therefore will be present in the scene.

3 Three.js's mechanics

3.1 Intersection

For the intersection between objects are used the box3 that rapresents an axis-aligned bounding box in 3d space. At each frame the room in which the animal is located is estimated and if there is also an obstacle in this, the intersection is estimated. The boxes3 are computed for each node of the obstacle, while the animal computed the boxes3 of its own body and head. The intersect method is then used for box3 of the head and box3 of the body with each box3 of the obstacle. If one of the boxes3 thus obtained is not empty, an intersection has occurred

3.2 RayCaser

The raycaster is used for character choice on the home screen. The raycaster throws a ray from a position and returns all objects intersected by the ray. Layers can be set so that the raycaster only interacts with objects with that particular layer. I then assigned the same layer to the raycaster and to the face and body of the animals.

3.3 Three animation system

With the exception of the rotation in the transition rooms, all animations are made through the three.js animation system.

3.4 Instanced Mesh

In three js if some objects have the same geometry and the same material but different transformation matrices these can all be instantiated through the intancedMesh(count) , with count equal to the number of objects we want to instantiate. Once the objects are instanced with the instancedMesh we can access them through the index and change their transformation matrix, one by one. The advantage of doing this is that they will be treated as a single mesh and therefore as a single draw call. For a matter of timing this feature was not used to the maximum.
It was used for the windows, always present in groups of 4 for each wall. Each window consists of two meshes for a total of 8 wall meshes. Using 2 instanced meshes, one for each window mesh, I reduced the draw calls from 8 to 2, which for cases where there are four walls in the scene, each with windows, means

having 8 (2 InstancedMesh * 4 walls) draw call instead of 32 (8 meshes * 4 walls). Although it is a consistent reduction in game draw calls amount to a much higher number, around 150, so this reduction has little impact.

4 Optimization

The main optimization was carried out on the triangles present in the scene. A reduction of these has been made by changing all the models so that only the parts of these that would have been visible by the player were actually rendered. This was possible through a rewrite of all models where the invisible parts are not created. We have thus gone from a number of triangles with peaks of 8000 to a smaller number with maximums of 2,000 triangles.
The second, more contained, optimization concerns draw calls as previously written.
The third concerns the use of geometries and materials. Since series of different objects (rooms, lights, obstacles) are created, geometries and materials are created only once and then used whenever an object needs to be meshed.