Simulation & Animation - SS 2022

Fury Road
Final Game

Miha Kosi, 12110733, mkosi@student.tugraz.at
Ekaterina Baikova, 12045991, ekaterina.baikova@tugraz.at

June 12, 2022

Game Manual

About the game

Fury Road is an adrenaline-filled racing game with great effects and graphics, innovative game play mechanics and good old fun.

You have found yourself on an old racing track. It is filled with smaller and larger obstacles and intersected by two circular railway lines with a train driving on each of them. Your mission is to follow the track and finish the lap in one piece as fast as possible.

Game over

The game ends when your car finishes the lap by crossing the finish line. When you finish the lap, your total lap time is displayed.

You can also fail to finish the game if your car is destroyed as a result of a collision with a train or too many collisions with obstacles on the racing track.

Interface

- **Lap time** is located in the upper right corner. It displays the elapsed time from the start of the lap.
- **Chequered flag** denotes the end of the lap.

Controls

Command	Button
Accelerate	↑
Brake	↓
Steer left	←
Steer right	→
Pause/Resume game	P
Settings	ESC

Running the game

The game can be played in a web browser, preferably Mozilla Firefox. It requires Node.js to be installed. It has been tested with Node.js 16.14.0 which can be downloaded from `https://nodejs.org/en/download/`. The game has not been tested with other versions of Node.js and may not be compatible with them.

To install the required packages, navigate to the root directory with the game source code (the one that contains `server.js`) and run **npm install** to install the required packages. When the packages are installed, run **node server.js** in the same directory to start a local server. The game should be available at `http://localhost:8080`.

Technical Documentation

General Settings

General settings simultaneously impact multiple techniques. The following settings can be changed:

- **Maximum frame rate** limits the maximum number of frames that are rendered every second. It can be set to 30 or 60.
- **Animation update rate** defines the factor of how much time passes between two frame updates.
- **Link animation update rate and animation speed** can enable or disable the reflection of animation update rate in animation speed. If disabled, the speed of the animation will always remain the same, regardless of how much time will elapse between two individual frame updates. If enabled, increasing or decreasing animation update rate will increase or decrease the speed of the animation.

Path Interpolation

The game features a train moving along a circular Catmull-Rom spline. The train first accelerates, then moves at a constant speed and finally decelerates, and after that the animation repeats itself indefinitely.

Path interpolation is implemented in *PathInterpol.js* script. For each of the control points, we compute the arc length table according to the chosen precision (in our game 0.05) which is implemented in *#interpolate* function. When the animation is started, we calculate the amount of distance the object should travel according to its defined speed and time that has passed between the previous and current frame and divide it by the number of control points so that the object does not move too fast. Based on the distance the object should travel and the arc length table that we had built before, we then calculate the parametric value at which we sample the curve, and this is the position the object is then placed at. The animation is ease in/ease out, and the only addition is that the distance is recalculated according to the *#ease* function prior to calculating the parametric value in *#animate* function.

Control points, interpolated points that represent arc-length table and calculated spline curve can all be visualised by toggling the corresponding setting in the settings menu. Traversal speed of the object that moves along the spline can also be changed in settings, as well as the animation update rate.

Rigid Body Dynamics

The game features objects on the side of the racing track, such as stones, buildings, tires and barrels that react to the collision with a player's car and with each other. All these objects have a solid body. Rigid Body is implemented in *RigidBody.js*. Rigid Body is a class that we create, that takes in the constructor the image of an object, its mass and maximum velocity that it can reach. Also, it has a static variable with all the *allRigidBodies* objects for detecting collision later.

The function *rectsIntersect()* checks if two rectangular bounding boxes of objects are intersecting via x and y coordinates of two bounding boxes with the anchor set in the centre of the rectangle. Then if the objects collide, the function *collisionVector()* calculates the vector between centres of objects, distance and normalised collision vector to get relative velocity.

We compute the result of collision in the centre mass frame of reference using relative velocity

$$x : obj1.vel_x - obj2.vel_x, y : obj1.vel_y - obj2.vel_y$$

and using energy and momentum conservation laws. Relative velocity is projected on collision vector to extract part of velocity that is used in momentum conservation: *speed*. For the angular momentum we use rotational energy and angular momentum conservation laws.

By toggling the corresponding settings in settings menu, momentum vectors and collision points can be visualised.

Motion Blur

The game features a train potentially moving at a very high speed. Since the object is moving faster than the individual frames can be rendered, the motion is blurred to produce an effect of a fast moving object.

Motion blur is implemented in *MotionBlur.js* script. Post-processing motion blur is implemented by averaging the colours of the pixels along the line of the velocity vector, moving by a maximum of one pixel in each step, and is done by using a filter. Supersampling motion blur is implemented by multiplying and drawing the object multiple times at equal intervals between the previous and current point and the previous and current rotation of the object. The opacity of each object in supersampling is $\frac{3}{N}$, where N is the number of samples, which resulted in best visuals. Using $\frac{1}{N}$ makes the objects too transparent, as stacking semi-transparent objects does not sum to the final opacity of 1, because less and less light comes through the objects as it progresses through the stacked objects.

Motion blur is always visible. By default post-processing motion blur is visualised, but can be switched to supersampling in game settings.

Voronoi Fracture

The game features many additional solid objects besides the players car itself. If a player's car collides with any of them, it is shattered using Voronoi fracture. Depending on how many times the collision was registered, the Voronoi fracture breaks the sprite into parts representing cracks on the car's surface.

The Voronoi fracture in *app.js* script: When global variable *collision count* reaches a certain number, the first container of Voronoi fracture is created to be displayed over player's car sprite with the same position and rotation. Every next Voronoi container represents the previous shattering but with 3 more seed points added. Overall, there are 3 stages of Voronoi fracture, the 4th stage is considered as a "game over" condition.

Functional part of Voronoi fracture in *Voronoi.js* : First, we generate initial seed points using sprite width and height and the number of points and put them into the container. Then for every pixel we find the two closest seed points and save their coordinates. After this, we compute distance D of pixel to the voronoi edge of two closest seed points using formulas from the lecture slides in *drawVoronoiCell()* and check if the seed point is the closest one, if it is the case we return -D and assign a pixel to a seed point's voronoi cell. Then we update "mask sectors" which was filled with zeros with *createMaskSectors()* functions, so that it contains the actual sectors for drawing them later. In *createEdgeArray()* we look for a differences between element and its right neighbour and the lower one. If it is not zero, we make an edge point to display it later.

By toggling the corresponding settings in settings menu, seed points and cell fields can be visualised.

Art References

- barrel.png
- car2.png
- train.png
- tire.png

Self-created assets:

- gas-station.png
- car1.png
- structure-1.png
- structure-2.png